Scheduling Techniques for Efficient Execution of Stream Workflows in Cloud Environments

by

Mutaz Barika

BSc. in Computer Science, September 2007, Petra University, Jordan

MSc. in Computer Science, June 2011, King Saud University, Kingdom of Saudi Arabia

School of Technology, Environments and Design — College of Sciences and Engineering

Submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

University of Tasmania February 2020

Declaration of Originality

This thesis contains no material which has been accepted for a degree or diploma by the University or any other institution, except by way of background information and duly acknowledged in the thesis, and to the best of my knowledge and belief no material previously published or written by another person except where due acknowledgement is made in the text of the thesis, nor does the thesis contain any material that infringes copyright.

Authority of Access

This thesis may be made available for loan and limited copying and communication in accordance with the Copyright Act 1968.

Statement regarding published work contained in thesis

The publishers of the papers comprising Chapters 2, 3 and 4 hold the copyright for that content, and access to the material should be sought from the respective journals. The remaining non published content of the thesis may be made available for loan and limited copying and communication in accordance with the Copyright Act 1968.

Mutaz Barika

15/02/2020

Statement of Co-Authorship

(for include in the thesis)

The following people and institutions contributed to the publication of work undertaken as part of this thesis:

Name and School = **Mutaz Barika, School of Technology, Environments and Design (Candidate)**
Name and institution, Supervisor = **Saurabh Garg, University of Tasmania (Co-Author 1)**
Name and institution, Co-Supervisor = **Andrew Chan, University of Tasmania (Co-Author 2)**
Name and institution = **Albert Y. Zomaya, University of Sydney (Co-Author 3)**
Name and institution = **Lizhe Wang, China University of Geoscience (Wuhan) (Co-Author 4)**
Name and institution = **Aad van Moorsel, Newcastle University (Co-Author 5)**
Name and institution = **Rajiv Ranjan, China University of Geoscience (Wuhan) and Newcastle University (Co-Author 6)**
Name and institution = **Rodrigo N. Calheiros, Western Sydney University (Co-Author 7)**

Author details and their roles:

Paper 1, Mutaz Barika, Saurabh Garg, Albert Y. Zomaya, Lizhe Wang, Aad van Moorsel and Rajiv Ranjan, "Orchestrating Big Data Analysis Workflows in the Cloud: Research Challenges, Survey, and Future Directions", ACM Computing Surveys, Volume 52 Issue 5, Article No. 95:
Located in Chapter 2
Candidate was the primary author and contributed approximately 80% to planning, conducting and reporting the research.
Co-Author 1, Co-Author 5 and Co-Author 6 contributed approximately 5% each to critical revision of the paper.
Co-Author 3 and Co-Author 4 contributed approximately 2.5% each to the refinement of the paper.

Paper 2, Mutaz Barika, Saurabh Garg, Andrew Chan, Rodrigo N. Calheiros and Rajiv Ranjan, "IoTSim-Stream: Modelling stream graph application in cloud simulation", Future Generation Computer Systems, Volume 99, 2019, pp. 86-105.:
Located in Chapter 3
Candidate was the primary author and contributed approximately 80% to designing and implementing IoTSim-Stream, conducting experiments, analysis of results and writing the paper.
Co-Author 1, Co-Author 2, Co-Author 6 and Co-Author 7 contributed approximately 5% each to critical revision of the paper.

Paper 3, Mutaz Barika, Saurabh Garg, Andrew Chan and Rodrigo N. Calheiros, "Scheduling Algorithms for Efficient Execution of Stream Workflow Applications in Multicloud Environments", IEEE Transactions on Services Computing, Accepted:
Located in Chapter 4
Candidate was the primary author and contributed 80% to designing and implementing scheduling algorithms, conducting experiments, analysis of results and writing the paper.
Co-Author 1 contributed approximately 10% to the development of the idea and critical revision of the paper.
Co-Author 2 contributed approximately 5% to the refinement and presentation of the paper.
Co-Author 7 contributed approximately 5% to the formulation of the idea and proofread the paper.

Paper 4, Mutaz Barika, Saurabh Garg, Albert Y. Zomaya and Rajiv Ranjan "Online Scheduling Technique To Handle Data Velocity Changes in Stream Workflows", IEEE Transactions on Parallel and Distributed Systems, under review:

Located in Chapter 5

Candidate was the primary author and contributed 85% to designing and implementing scheduling technique, conducting experiments, analysis of results and writing the paper.

Co-Author 1 contributed approximately 10% to the refinement of the idea and critical revision of the paper.

Co-Author 3 and Co-Author 6 contributed approximately 2.5% each to the refinement of the paper.

Paper 5, Mutaz Barika, Saurabh Garg and Rajiv Ranjan "Cost Effective Stream Workflow Scheduling To Handle Application Structural Changes", Future Generation Computer Systems, under review:

Located in chapter 6

Candidate was the primary author and contributed 85% to designing and implementing scheduling techniques, conducting experiments, analysis of results and writing the paper.

Co-Author 1 contributed approximately 10% to the refinement of experimental design and critical revision of the paper.

Co-Author 6 contributed approximately 5% to the refinement and presentation of the paper.

We the undersigned agree with the above stated "proportion of work undertaken" for each of the above published (or submitted) peer-reviewed manuscripts contributing to this thesis:

Signed:

Dr. Saurabh Garg
Supervisor
School of Technology, Environments and Design
University of Tasmania

Date: 7/02/2019

Professor Elaine Stratford
Head of School (acting)
School of Technology, Environments and Design
University of Tasmania

13/2/2020

Acknowledgements

I would like to express my sincere thanks to my primary supervisor Dr. Saurabh Garg for providing me the opportunity to pursue my PhD and supervised me with dedication over the years. I deeply appreciate your patience and continuous support to push me to be a better researcher. I must also thanks Prof. Andrew Chan, my co-supervisor, for his support and advice throughout my PhD. Thank you, my supervisors, for your guidance and insightful comments that are essential to complete my thesis.

I would also like to thank Prof. Rajiv Ranjan, Prof. Albert Y. Zomaya, Prof. Lizhe Wang, Prof. Aad van Moorsel and Dr. Rodrigo N. Calheiros for their knowledge as co-authors in my publications in this thesis. I extend my appreciation to Prof. Rajiv Ranjan for his support and advice on all things. I would like also to acknowledge Prof. Rajkumar Buyya for the insightful comments and suggestions in improving the quality of Chapter 4.

I thank Alexander Brown for proof-reading the introduction and conclusion of my thesis.

Lastly and most importantly, I express my sincere gratitude to my parents Sadka and Seleam for their endless efforts to support me throughout my many years of study. My endless gratitude to my mother, who has passed away when I was at the end of my second year, but all her support and encouragement are with me till today, so that I achieve what I have today. A special big thank you to my big brother Foad for supporting me not only during my PhD, but in my whole life. I really cannot thank you enough for everything you have done for me from your help in my hard times to your sacrifices, putting all your efforts to make the person that I am today. Thank you all for supporting me to achieve my dreams.

This research is supported by an Australian Government Research Training Program (RTP) Scholarship.

Abstract

Advancements in Internet of Things (IoT) technology have led to the development of advanced applications and services that rely on data generated from enormous amounts of connected devices such as sensors, mobile devices and smart cars. These applications process and analyse such data as it arrives to unleash the potential of live analytics. Considering that our future world will be fully automated, current IoT applications and services are categorised as data-driven workflows, which integrate multiple analytical components. Examples of these workflow applications are smart farming, smart retail and smart transportation. This workflow application also known as a stream workflow is one type of big data workflow application and is becoming gradually viable for solving real-time data computation problems that are more complex.

The use of cloud computing technology which can provide on demand and elastic resources to execute stream workflow applications is ideal, but additional challenges are raised due to the location of data sources and end users' requirements in terms of data processing and deadline for decision making. The focus of existing research works in this domain is on the streaming operator graph generated by streaming data platforms, where this graph differs from a stream workflow as there is a single source of data for the whole operator graph and one end operator, while stream workflow has multiple input data sources and multiple output streams. Moreover, the majority of those works investigated one type of runtime change for the streaming graph operator, which is the fluctuation of data. This means that the structural changes that may happen at runtime are not studied. Considering the heterogeneity and dynamic behaviour of stream workflows, these workflow applications have unique features that make the scheduling problem have different assumptions and optimisation goals compared with the placement problem of streaming graph operators.

As a consequence, the execution of stream workflow applications on the cloud environment requires advanced scheduling techniques to address the aforementioned challenges as well as handling different runtime changes that may occur during the execution of these applications. To this end, the Multicloud environment approach opens the door toward enhancing the execution of workflow applications by leveraging various clouds to utilise data locality and exploit deployment flexibility. Thus, the problem of scheduling a stream workflow in a Multicloud environment while meeting user real-time data analysis requirements needs to be investigated. In this thesis, we leverage the Multicloud environment approach

to design novel scheduling techniques to efficiently schedule outsourcing stream workflow applications over various cloud infrastructures while minimising the execution cost. We also design dynamic scheduling techniques to continuously manage resources to handle structural and non-structural changes at runtime in order to maintain user-defined performance requirements at minimal execution cost. In summary, this thesis makes the following concrete contributions:

- Comprehensive state of the art survey that analyses various big data workflow orchestration issues span over three different levels (workflow, data and cloud) by providing a research taxonomy of core requirements, challenges, and current tools, techniques and research prototypes.
- Simulation toolkit named IoTSim-Stream to model and simulate stream workflow applications in cloud computing environments.
- Two scheduling algorithms that generate scheduling plans at deployment time to execute stream workflow efficiently on cloud infrastructures with minimal monetary cost.
- Two-phase adaptive scheduling technique that considers the problem of scheduling stream workflows to support runtime data fluctuations while guaranteeing real-time performance requirements and minimising monetary cost.
- Pluggable dynamic scheduling technique that manages cloud resources over time to handle structural changes of stream workflow at runtime in a cost-effective manner, along with three plugin scheduling methods.

Contents

Chapter 1

Introduction

1.1 Research Background

Recently, Internet of Things (IoT) has been going mainstream and shaping the future of connectivity (Taneja and Davy, 2017). Under the umbrella of the IoT ecosystem, everything should be connected to the Internet. This connectivity enables communication and data exchange among connected things, and it exhibits massive amounts of streaming data (Sowe et al., 2014) (Zomaya and Sakr, 2017). The continuous growth of IoT technology makes the size of big data larger than we previously thought. Specifically, IoT is the largest source of big data (Sowe et al., 2014) (Mohammadi et al., 2018). Certainly, the more data we have, the more problems we need to address. This raises a high demand for quickly processing and analysing IoT big data to derive real-time analytics and turn them into immediate actions (Mohammadi et al., 2018). Stream processing is the key enabler for real-time analysis that opens the door for developing future intelligent and pervasive IoT analytics platforms (Tönjes et al., 2014) (Ge et al., 2016) (Amini et al., 2017). With the interest of real-time data analytics, different streaming big data applications have been constantly developed to process IoT data as it becomes available. Toward a smart world with smart things, these applications are integrated into a data analysis pipeline forming a stream workflow. This workflow application is one type of big data workflow application and is categorised as a data-driven workflow. It is becoming gradually viable for solving real-time data computation problems that are more complex.

As a demonstration of a real use case for a stream workflow application that shows the need for real-time analytics and workflow scheduling and execution in the next era of technologies, consider connected vehicles in smart cities. Since traffic is strained with a continued increase of the number of vehicles and population, the smart road traffic monitoring as one of many smart city services can utilise the true power of IoT connected vehicles in addition to roadside infrastructure (e.g. traffic lights, cameras). Collecting and analysing the streaming data generated by these vehicles allow to create a real-time view of road traffic and incidents. Figure 1.1 depicts a streaming data pipeline for generating a real-time view of

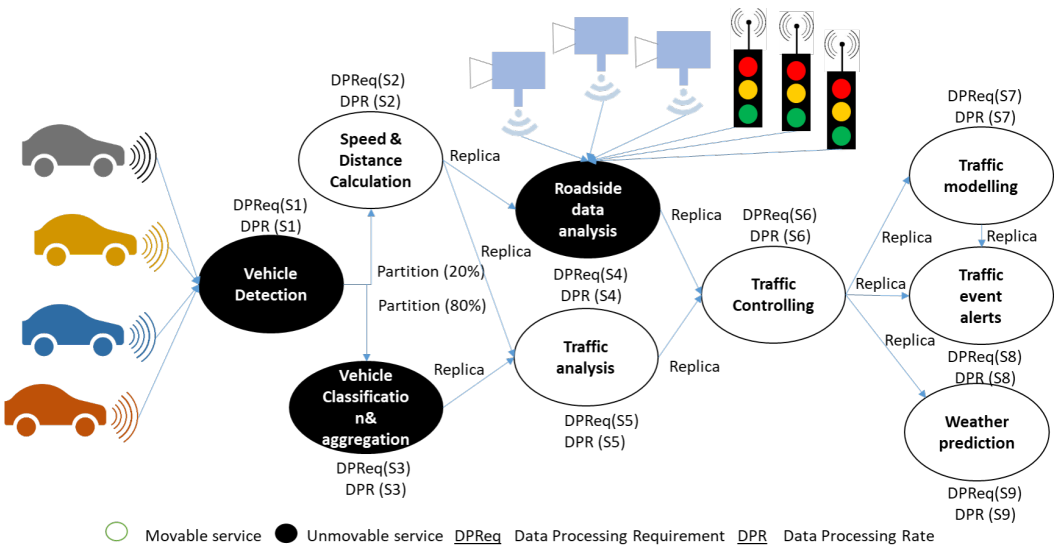


Figure 1.1: Dynamic stream workflow application example

road traffic in a smart city. This pipeline represents a dynamic stream workflow application, which is a network of streaming analytical components. Each analytical component can be seen as a service because it can independently execute over any virtual resources, even though data dependencies among services should be maintained.

In the presented workflow application, streaming data generated by connected vehicles on the road as sensor data is injected into the vehicle detection service to identify the presence of vehicles in real-time. This service produces continuous output streams that are partitioned between two services for applying specific calculation, and classifying and aggregating vehicle data. The information of movement flow which is based on vehicle speed and distance as a result of continuous computations, is further analysed by roadside data analysis and traffic analysis services. Based on this movement flow and sensor data coming from traffic lights and cameras, the roadside data analysis service produces real-time information about traffic density, which gives the opportunity to adjust traffic patterns. Also, the traffic analysis service generates in-depth real-time analytics for traffic patterns and conditions by processing and analysing two data stream inputs, the movement flow information and the aggregated vehicle data. These analytics are injected as output streams into the traffic controlling service for further processing to improve traffic modelling, alert road users and authorities about traffic events and predict the current weather.

The execution and management of a stream workflow application cannot be handled by traditional distributed systems because they experience several issues such as high scalability costs, resource utilisation and performance (Chang et al., 2005) (Yuan et al., 2012). Thus, a dynamic environment that is capable of providing a large of distributed resources is needed to meet real-time data analysis requirements. Because cloud computing provides on demand resources including compute, storage and network (Höfer and Karagiannis, 2011) (Ranjan et al., 2017), it is seen as a visible solution for running this application. Moreover,

the Multicloud environment, which consolidates multiple cloud infrastructures, opens the door toward enhancing the execution of stream workflow applications by carrying out this execution in a distributed manner, making it an ideal execution environment for this application. However, the distrusted workflow execution, as pointed in (Barika et al., 2019c), is a complicated process which poses several challenges such as ensuring low latency and reducing data transfer cost. Furthermore, the complexity and dynamism of a stream workflow raise additional challenges due to the following reasons:

- The continuous running of each analytical component.
- The velocity of data which depends on various factors such as data processing rate of parent component and network performance.
- The location of data source
- User-defined real-time performance requirements.

These challenges along with the limitations of existing research works to schedule and execute stream workflow in cloud computing are discussed in Section 1.3.

1.2 Research Problem

Based on the above analysis, our research problem in this thesis is *how to coordinate the execution of stream workflow application across multiple clouds to improve execution performance with minimal costs while adhering to user performance requirements?*

From the main research problem, our research questions that we will be investigated in this thesis are as follows:

1. What is the landscape of big data analysis workflow management and research gaps?
2. How to model stream workflow application based on user performance requirements?
3. How to find an optimal or near-optimal schedule and resource allocation to efficiently execute stream workflow application in cloud infrastructures?
4. How to maintain an efficient schedule of a stream workflow application under its various dynamic aspects at runtime?

1.3 Research Challenges

The focus of existing research works (such as (Berthold et al., 2005), (Pietzuch et al., 2006), (Khandekar et al., 2009) and (Carbone et al., 2015)) is on the streaming operator model employed by several streaming data platforms such as Apache Storm [1] and Apache Flink

[1] https://storm.apache.org/

[2], forming a streaming operator graph to extract fresh data analytics from infinite data sequences. This operator graph represents a data pipeline that involves a set of operators and data stream flows (i.e. edges carry data from parent operator to child operator), where there is one feeding source for the whole graph and one sink operator. Research works (Pietzuch et al., 2006), (Cardellini et al., 2016) and (Venkataraman et al., 2017) addressed the placement problem for operator graphs in distributed large-scale environments with various limitations that vary between them such as data locality unutilised or limited consideration for user-oriented QoS attributes. However, stream workflows are more complex and heterogeneous, involving analytical components that have heterogeneous user, platform and infrastructure requirements and these components are always in an active state. They also have multiple data sources that inject their data streams into any analytical components and have multiple outputs. Simply, the streaming graph operator is considered a simplified version of the stream workflow. Therefore, we require research into new scheduling technique that consider the new characteristics of stream workflow to execute them efficiently in cloud infrastructures.

Moreover, the velocity of data may fluctuate over time when deploying the streaming operator graph in a dynamic environment. This variation in data rate is considered as one type of runtime change for the streaming graph operator that has been investigated by the majority of existing studies in the literature. Research works (Sun et al., 2015), (Buddhika et al., 2017) and (Bożek and Werner, 2018) addressed the scheduling problem of data stream computations with the aim to improve performance and/or reduce energy consumption, while (Sun and Huang, 2016) and (Sun et al., 2018) addressed the problem of scheduling big data streaming application with the aim of guaranteeing makespan and utilising single cloud as an execution environment. Also, several streaming analytics systems and cloud-based services for real-time analytics such as Apache Storm, Microsoft Azure Stream Analytics and IBM Streaming Analytics, allow to create an operator graph that analyses data streams to produce final output stream. However, all of these research works and systems consider the simplified versions of the stream workflow (i.e. operator graph or big data stream application which is not a workflow of workflows). As a stream workflow is different to a streaming operator graph, those research works addressed the dynamic scheduling problem for different workflow application models, so that they have different optimisation goals. Therefore, there is a need for a new dynamic scheduling technique to deploy a stream workflow efficiently in a Multicloud environment and handle the change in velocity of data by revising a scheduling plan at runtime as quickly as possible and in a cost-effective manner.

Furthermore, the fluctuation of data in stream workflows is not the only change that may happen at runtime. The changes to the structure of stream workflows are of the utmost importance to be handled during the execution of these workflows. This is because these changes reflect the new amendments to control and/or data flows. The existing research

[2]https://flink.apache.org/

works (Liu et al., 2016b), (Liu and Buyya, 2017), (Kombi et al., 2019), (Sun and Huang, 2016) and (Sun et al., 2018) are focused on the streaming graph operator as well as they lack the ability to deal with application-level changes that may occur at runtime. Thus, the problem of handling structural changes that happen at runtime are not studied in the literature. Considering the heterogeneity and dynamic behaviour of stream workflows, these workflow applications have unique features that make the scheduling problem have different assumptions and optimisation goals compared with the placement problem of streaming graph operators. Therefore, there is an additional limitation that needs to be addressed. This gap requires research into new dynamic scheduling techniques that deal with various structural changes and their consequences.

The existing big data workflow orchestration systems are Apache Yet Another Resource Negotiator (YARN) and Apache Mesos. Apache YARN (Vavilapalli et al., 2013) (Apache, 2017) is a cluster resource management technology that schedules jobs and manages resources in the cluster. Apache Mesos (Hindman et al., 2011) (Sphere, 2017a) is an open-source cluster manager that abstracts the entire datacentre and shares the clustered resources among distributed applications. These systems employed a fair sharing model to equally share the resources of a cluster among applications over time. However, Apache YARN and Apache Mesos focused on supporting stream processing through operator graph or do not need to meet real-time user requirements. Also, they are not able to handle different dynamic forms (structural and non-structural changes) of stream workflows by managing the resources at runtime.

To overcome the aforementioned limitations, we propose different scheduling techniques that ensure the efficient mapping of analytical components involved in stream workflows to resources from multiple cloud infrastructures and manage these resources over time to handle runtime changes in a cost-effective manner while guaranteeing user real-time data processing requirements. These techniques address both static and dynamic scheduling problems of stream workflows in cloud environment, providing a complete scheduling solution that supports the entire scheduling process for this workflow application.

1.4 Research Objectives

The objectives of this thesis study are as follows:

1. Surveying big data workflow orchestration in the cloud.
2. Modelling stream workflow applications and their behaviours in a cloud computing environment.
3. Achieving efficient execution of stream workflow applications.
4. Maintaining efficient runtime scheduling of stream workflow applications.

1.5 Research Methodology

In order to answer the outlined research questions and satisfy the objectives of thesis, our research approaches are as follows:

- Survey and taxonomic approach − For any domain, reviewing the extant literature to formulate the problem being studied is the most popular methodological approach. It helps to uncover answers for research questions being identified and discusses the key issues that not yet resolved. Therefore, we conduct a comprehensive state of the art review and provide a research taxonomy to capture the landscape of big data workflow orchestration in the cloud.

- Multicloud environment approach − For running stream workflow application in cloud, there are two approaches: single cloud and Multicloud. With the single cloud environment approach, the whole execution of stream workflow application is bonded on resources provisioned from this cloud, which makes meeting user requirements is challenging because of the distribution of data sources. This means that if not all data sources that inject their data streams into a workflow are near to that cloud, the cost of transferring huge amounts of data is expensive and latency will be significant, leading to a delay in real-time data analysis and degrade overall performance. Therefore, it is necessary to utilise data locality by using a workflow model that targets distributed data sources by leveraging a Multicloud environment. In addition to utilising data locality, the Multicloud environment approach provides the flexibility to deal with changes in the location of data source over time, thus the analytical component deployed in a particular cloud can be redeployed to another cloud infrastructure according to the new location of data source. By doing this, transferring data over long-heal networks with expensive cost and latency is avoided. Furthermore, the flexibility of deploying analytical components in any cloud and changing their placement open different areas for improvement. For example, reducing deployment cost by selecting the cheapest cloud instances or improving performance by distributing data processing workloads. For our research problem in this thesis, leveraging the power of the Multicloud environment allows to distribute the workloads among different clouds to meet Service Level Agreement (SLA) and Quality of Service (QoS) needs of stream workflows. Thus, it is an ideal execution environment for these workflows as it not only facilitates meeting real-time data analysis requirements of such applications, but helps to improve performance, minimises the execution cost and deals with runtime changes in a cost-effective manner. In other words, it paves the way toward enhancing the execution of stream workflow applications and dealing with its dynamic characteristics by making the use of various clouds to utilise data locality and exploit deployment flexibility. Accordingly, we leverage the Multicloud environment to distribute analytical components involved in stream workflow application among multiple clouds for efficient execution.

- Simulation environment approach − A simulated environment helps researchers to model their applications, study how these applications will behave and evaluate the performance of the modelled applications in a controllable environment. In the cloud computing domain, many research works have proposed simulation toolkits to model and study different types of applications such as CloudSim (Calheiros et al., 2011) for cloud computing infrastructures, applications and services, NetworkCloudSim (Garg and Buyya, 2011) for cloud datacenter network and parallel applications, IoTSim (zen,) for IoT applications with MapReduce model, CEPSim (Higashino et al., 2016) for event processing queries, and WorkflowSim (Chen and Deelman, 2012b) for scientific workflow applications. For stream workflow applications, the use of large, heterogeneous and distributed cloud platforms is too complicated for studying the behaviour of these applications and evaluating their performance. This is because the evaluation study on these platforms are subject to the impact of external events, notably not cost-effective, considerably time-consuming and different conditions cannot be reproducible to easily reproduce results. Also, the complexity of stream workflow and its requirements along with its dynamic characteristics make completing the extensive performance evaluations in real environment with different parameters and different settings far from being achieved. Accordingly, we follow a simulation methodology and propose a new simulation toolkit for studying the behaviour of stream workflow applications in repeatable, controllable, dependable and scalable environments.

- Scheduling and resource management approach − There are multiple levels at which performance can be improved, for example, optimisation at task / software-level, scheduling-level or hardware-level. However, in cloud computing, effective workflow scheduling can help to improve the efficiency of workflow execution. As such various workflow scheduling approaches have been proposed by researchers such as (Chen and Deelman, 2011), (Fischer and Bernstein, 2015), (Zhu et al., 2016), (Rehani and Garg, 2017) and (Chen et al., 2018b). For stream workflow applications, there is a need to propose scheduling techniques to efficiently execute this application in cloud computing while adhering to user real-time data analysis requirements. The design of these techniques is challenging due to the complexity of stream workflow and real-time performance requirements (i.e. throughputs and response time). Therefore, new scheduling and resource allocation techniques leveraging the power of Multicloud environment are proposed in this thesis. The evaluation of the proposed techniques is carried-out in a simulation environment that is similar to a real-world environment. This is because we used real workflow structures based on stream workflow applications and real values for all experiment configurations obtained from IoT technologies/standards, and real cloud infrastructures with their network performance.

  In addition, a stream workflow is adaptive in that it serves future user demand for real-time data analysis in a highly dynamic environment, i.e. IoT. The amendments

in the structure of real-time data pipelines and the requirements of data analysis are no longer the exception, but rather occur to reflect live changes in this environment. To further enhance the performance of workflow execution and respond to runtime changes, scheduling-level optimisation can be performed dynamically. Thus, an elastic scheduling approach is used to manage resources over time to guarantee real-time data analysis requirements while handling different dynamic forms of a stream workflow. In this thesis, we design new dynamic scheduling techniques to support elasticity and amend scheduling plans to respond to runtime changes that happen during the execution of stream workflows. The efficiency evaluation of the proposed techniques is also carried out in a simulation environment.

1.6 Thesis Contribution

This thesis makes the following concrete contributions:

- Comprehensive state of the art survey that analyses various big data workflow orchestration issues spanning over three different levels (workflow, data and cloud) by providing a research taxonomy of core requirements, challenges, and current tools, techniques and research prototypes. It captures the research landscape of big data workflow orchestration in the cloud and highlights the key open issues in this area. This survey can be utilised by experts from both industry and research communities to derive further research. This contribution achieved the first objective and is detailed in Chapter 2.

- Simulation toolkit named IoTSim-Stream to model and simulate stream workflows on cloud infrastructures. This simulator integrates a stream processing model with workflow scheduling and execution. It can be utilised by researchers to easily setup a Multicloud environment and customise user real-time performance requirements to study the behaviour of stream workflow applications with different structures and configuration sizes. IoTSim-Stream is a simulation toolkit that deals with the complexities of modelling stream workflow applications and simulated environments. This contribution achieved the second objective and is detailed in Chapter 3.

- Two scheduling algorithms that generate scheduling plans at deployment time to execute stream workflow on cloud infrastructures with minimal monetary cost. The first algorithm is a greedy algorithm looking for local optimum while the other is a genetic algorithm looking for global optimum. This contribution achieved the third objective and is detailed in Chapter 4.

- Two-phase adaptive scheduling technique that considers the problem of scheduling stream workflows to support runtime data fluctuations while guaranteeing real-time performance requirements and minimising monetary cost. This technique incorporates

two advanced optimisation algorithms to efficiently execute this type of workflow application. The first algorithm is a random immigrants genetic algorithm that finds near-optimal solution in the complex search space at deployment time and the second one is a two-level greedy algorithm that amends the current scheduling plan at runtime to handle the change in data velocity. This contribution achieved the fourth objective and is detailed in Chapter 5.

- Pluggable dynamic scheduling technique that manages resources over time to handle stream workflow structural changes at runtime, achieving efficient execution of stream workflow in a cloud computing environment. It allows user to plugin her/his scheduling algorithms and methods without dealing with the complexity of scheduling processes and focusing only on the implementation of heuristic decisions. In addition, we proposed three different methods that can be plugged-in separately into the proposed technique to produce three different elastic scheduling techniques. These techniques are a baseline technique, a dynamic fair-share technique and a optimisation technique. This contribution achieved the fourth objective and is detailed in Chapter 6.

1.7 Thesis Organisation

The rest of this thesis is outlined in Figure 1.2 and structured as follows: Chapter 2 presents the current state of the art of orchestrating big data workflows in the cloud by providing research taxonomy and identifying key issues that not yet investigated in this area. In this chapter, distributed workflow execution is listed as one of the open issues, thus the need of modelling and executing stream workflow in the cloud is revealed. Chapter 3 first proposes a new IoT stream simulator (IoTSim-Stream) that is used in this thesis to model and simulate the execution of stream workflow applications in cloud environment. This chapter also presents the architecture of the proposed simulator with the details of design, implementation and evaluation. Then, Chapter 4 investigates the scheduling problem of stream workflow applications over multiple clouds and proposes two Multicloud scheduling algorithms to efficiently deploy and execute these applications based on real-time user performance requirements with minimal execution cost. Next, Chapter 5 discusses the dynamic scheduling problem of stream workflow applications over cloud infrastructures and how to handle data velocity changes. This chapter proposes a new dynamic scheduling technique to amend scheduling plan at runtime to cope with fluctuations of data. Chapter 6 further discusses the dynamic scheduling problem and models different dynamic forms of stream workflow application. This chapter proposes a scalable and pluggable dynamic scheduling technique to cope with different runtime changes that happen during the execution of stream workflow applications in the cloud. Last, Chapter 7 concludes the thesis and highlight future works.

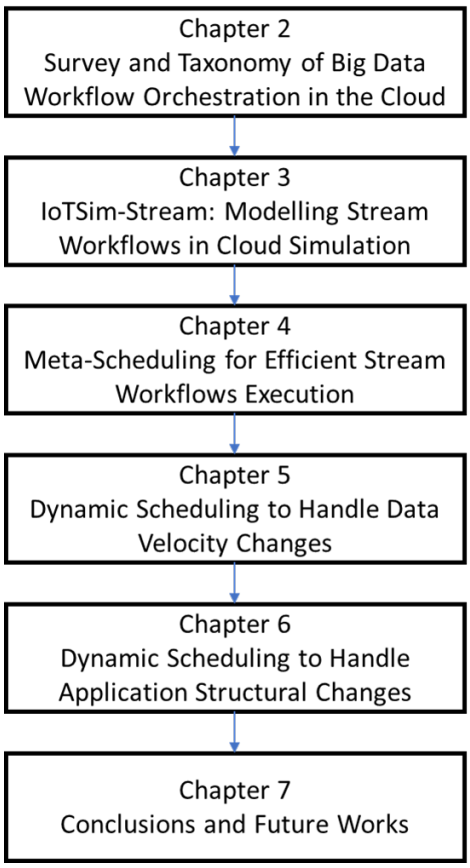


Figure 1.2: Thesis Organisation

1.8 Publication Record

The work presented in this thesis has been partially or completely published in the following set of publications:

- Mutaz Barika, Saurabh Garg, Albert Y. Zomaya, Lizhe Wang, Aad van Moorsel and Rajiv Ranjan, "Orchestrating Big Data Analysis Workflows in the Cloud: Research Challenges, Survey, and Future Directions", ACM Computing Surveys, Volume 52 Issue 5, Article No. 95. Chapter 2 is derived from this publication.

- Mutaz Barika, Saurabh Garg, Andrew Chan, Rodrigo N. Calheiros and Rajiv Ranjan, "IoTSim-Stream: Modelling Stream Graph Application in Cloud Simulation", Future Generation Computer Systems, Volume 99, 2019, pp. 86-105. Chapter 3 is derived from this publication.

- Mutaz Barika, Saurabh Garg, Andrew Chan and Rodrigo N. Calheiros, "Scheduling Algorithms for Efficient Execution of Stream Workflow Applications in Multicloud Environments", IEEE Transactions on Services Computing, Accepted. Chapter 4 is derived from this publication.

- Mutaz Barika, Saurabh Garg, Albert Y. Zomaya and Rajiv Ranjan, "Online Scheduling Technique To Handle Data Velocity Changes in Stream Workflows", IEEE Transactions

on Parallel and Distributed Systems, under review. Chapter 5 is derived from this publication.

- Mutaz Barika, Saurabh Garg and Rajiv Ranjan, "Cost Effective Stream Workflow Scheduling To Handle Application Structural Changes", Future Generation Computer Systems, Volume 112, 2020, pp. 348-361. Chapter 6 is derived from this publication.

Chapter 2

Survey and Taxonomy of Big Data Workflow Orchestration in the Cloud

Big data workflows consist of cross-disciplinary applications where timely results are critical: agriculture, transport, water management, healthcare, finance, utility networks and environmental monitoring. Stream workflow as one of big data workflow applications, integrates multiple streaming big data applications into data pipeline to support better decision making. Scheduling and executing stream workflows in the cloud requires deep and fundamental understanding of the existing challenges, approaches, techniques and tools in this field and related fields. Therefore, in this chapter, we discuss in detail the requirements for executing these workflows in the cloud as well as the challenges in achieving those requirements. We also present the state of the art, a provide research taxonomy and identify open research challenges in the area of study.

2.1 Introduction

The complexity of supporting big data analysis is considerably larger than the perception created by recent publicity. Unlike software solutions that are specifically developed for a specific application, big data analytics solutions typically require to integrate existing trusted software components in order to execute the necessary analytical tasks. These solutions need to support the high velocity, volume and variety of big data (i.e. 3Vs of big data (Liu et al., 2016a)) and thus should leverage the capabilities of cloud datacenter computation as well as storage resources as much as possible. In particular, many of the current big data analytics solutions can be classified as data-driven workflows, which integrate big data analytical activities in a workflow. Analytical tasks within these big data workflow applications may require different big data platforms (e.g. Apache Hadoop or Storm) as well as a large amount of computational and storage resources to process large volume and high velocity data. Intrusion detection, disaster management, and bioinformatics applications are some examples of such applications.

Big data workflows are very different from traditional business and scientific workflows (see Appendix A.2), as they have to continuously process heterogeneous data (batch and streaming data) and support multiple active analytical tasks at each moment in time. Moreover, they involve analytical activities that have heterogeneous platform and infrastructure requirements, and the overall workflows can be highly dynamic in nature, because processing requirements at each step are determined by data flow dependencies (the data produced in earlier steps in the workflow) as well as control flow dependencies (the structural orchestrations of data analysis steps in the workflow). In addition, big data workflows are different from streaming operator graphs formed by streaming processing systems like Apache Storm and Flink, as they have heterogeneous analytical activities, and involve multiple data sources that inject their data into upstream and/or downstream analytical activities and multiple outputs; but these systems employ continuous operator model to process streaming data only and have one cluster, and they form an operator graph with one feeding data source and one sink operator.

The focus of previous workflow taxonomy studies (Giaglis, 2001) (Yu and Buyya, 2005) (Rahman et al., 2011) (Poola et al., 2017) are on either business processes and information systems (for (Giaglis, 2001) and (Poola et al., 2017)) or Grid computing and its applications (for (Yu and Buyya, 2005) and (Rahman et al., 2011)). Given the advancement in big data applications and systems, new surveys are required that can synthesise current research and help in directing future research. Some recent surveys such as (Sakr et al., 2013), (Sakr et al., 2011) and (Mansouri et al., 2017) focused on a specific aspect of the big data applications and their management. They have not given overall dimensions involved in their orchestration such as workflow initialisation and parallelisation. For example, (Sakr et al., 2013) focused on big data analysis with MapReduce model research, while (Sakr et al., 2011) and (Mansouri et al., 2017) studied only data management aspects for deploying data-intensive applications

in the cloud. Recently, (Liu et al., 2018) provided a survey covering scheduling frameworks for various big data systems and a taxonomy based on their features without any in-depth analysis of issues related to big data analytical workflows and their orchestration.

However, big data workflow applications processing big data exhibit different patterns and performance requirements that traditional workflow processing methods and current workflow management systems cannot handle efficiently. Therefore, we require research into new orchestration models as well as orchestration platform and management technologies that can provide services to support the design of big data workflows, the selection of resources (including at platform and infrastructure), and the scheduling and deployment of workflows. These needs drive us to investigate the answer of the following research questions: (1) what are the different models and requirements of big data workflow applications?, (2) what are the challenges based on the nature of this type of workflow application and cloud + edge datacenters that we will face when developing a new big data orchestration system? and (3) what are the current approaches, techniques, tools and technologies to address these challenges?

To the aforementioned research questions, we present an exhaustive survey of big data programming models (see Section 2.2.3). We further elaborate on this survey to explain the relationship between big data programming models and workflows (Appendix A.1). We also propose a comprehensive research taxonomy to allow effective exploration, assessment and comparison of various big data workflow orchestration issues (see Section 2.4) across multiple levels (workflow, data and cloud). Moreover, we apply the proposed research taxonomy for surveying (see Section 2.5) a set of carefully chosen big data workflow orchestration tools (see Appendix A.3), orchestration techniques, and research prototypes. Furthermore, we identify current open research issues (see Section 2.7) in the management of big data workflows based on the literature survey and requirements.

This chapter is structured as follows: Section 2.2 presents a typical example of big data workflow that spans the three layers (workflow, data and cloud) and its orchestration in a cloud system. Section 2.3 highlights the key important requirements of big data workflows while in Section 2.4, we present a taxonomy for challenges in fulfilling those requirements. Section 2.5 presents the current approaches and techniques to address these challenges. Section 2.6 reviews scientific workflow systems with data-intensive capabilities and big data orchestrating systems, and discusses the capabilities of big data orchestrating systems against the presented research taxonomy. Section 2.7 presents and discusses the open issues for further research, while Section 2.8 concludes the chapter.

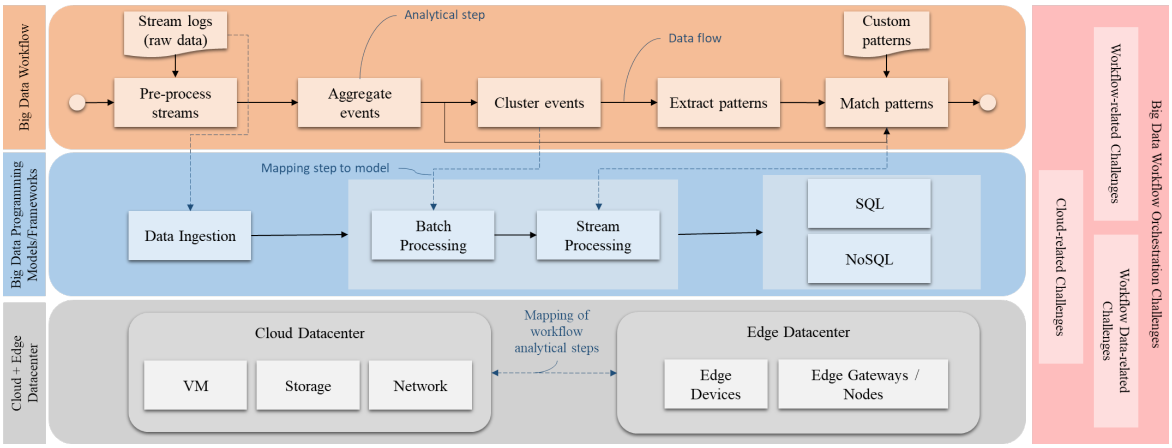


Figure 2.1: An example workflow for anomaly detection over sensor data streams and its mapping to programming models/frameworks and cloud + edge datacenters.

2.2 Big Data Workflow Orchestration

2.2.1 Representative Example of a Big Data Workflow

To aid understanding of big data workflows and the issue of orchestrating such workflow applications in the cloud and edge resources, we present a typical example of anomaly detection (shown in Figure 2.1). It is a representation of the workflow presented in (Ano, 2015).

The data pipeline is used to analyse sensor data streams for online anomaly detection. The representation of this workflow spans the three layers (workflow, data and cloud) is shown in Figure 2.1. First of all, the streams of data (i.e. stream logs) are ingested in the pipeline by following a message ingestion model (i.e. Kafka), where all events that are collected within a window of time are pre-processed by filtering and enriching them with additional metadata, e.g. external timestamps. Next, aggregation of events is done, for example per region or sensor type in a given window of time, which get clustered into different categories and later passed to pattern matching step (last step). At the cluster events step, a clustering-based outlier detection algorithm will run in a batch fashion over all produced aggregated events in order to generate outliers (possible/proposed anomalies). After that, all outliers are mined to extract possible frequent patterns, and those extracted patterns are further transformed into complex event processing queries reliant on the selected strategy. Finally, all patterns are matched to output the outliers by constantly injecting the rules into distributed complex event processing engines, and these engines perform continuous queries on the streams of data coming from either the pre-processing step or aggregate step for online anomaly detection. Accordingly, the stream programming model is followed for processing and analysing sensor data streams ingested in this workflow to produce continuous insights (online anomaly detection) by using Apache Storm (Ano, 2015); also, the anomaly patterns and analysis results generated in this workflow could be stored in SQL or NoSQL databases.

From the above example, we can easily see that the analytical tasks included in the data pipeline require seamless coordination for real-time and dynamic decision making hand-

ling different types of heterogeneity and uncertainties such as changing in data velocity or data volume. That includes (1) fulfilling the need of diverse computational models for pre-processing streams, aggregating and clustering events, and extracting possible frequent patterns; (2) managing inter-dependent analytical tasks, where any change in execution and performance characteristics of one can affects the downstream steps; and (3) matching patterns' analytical task need to take the advantage of edge resources available at edge datacenters to perform edge analytics, avoiding any possible latency. Therefore, to achieve this seamless execution for such types of workflow, various programming tasks needs to be performed, leading to several challenges related to cloud + edge resources and data orchestration, which span over three different levels (workflow, data and cloud).

2.2.2 Workflow Level

One of the aims of the big data workflow orchestration platform is to manage the sequence of analytical tasks (formed workflow application) that needs to deal with static as well as dynamic datasets generated by various data sources. This includes various programming tasks i.e. workflow composition and workflow mapping (Ranjan et al., 2017). Workflow composition is to combine different analytical tasks, where their workloads are dependent on each other and any change made in the execution and characteristics of one step affects the others. Therefore, different users of the workflow define their requirements and constraints from different contexts, resulting in different analytical tasks of a workflow needing to be executed, where the requirements are not only different but may also conflict with each others. Accordingly, a workflow orchestration system should provide the guidance for domain experts to define and manage the entire pipeline of analytical tasks, data flow and control flow, and their SLA and QoS needs. It can support different workflow orchestration techniques to compose heterogeneous analytical tasks on cloud and edge resources including script-based (that defines composition flow using script languages), event-based (that uses event rules defined in workflow language to provide responsive orchestration process) or adaptive orchestration (that dynamically adopts composition flow in accordance to application and execution environment needs). IoT and Cyber-Physical Systems (CPS) have several requirements such as processing a large amount of data streams from physical world, the real-time decision making to sensor events and dynamic management of data flow (Seiger et al., 2018). In other words, IoT and CPS applications are adaptive workflows involving event-driven tasks that sophistically analyse data streams for decision making. These workflows can be considered as a specific exemplar of big data pipeline and managed under the umbrella of big data workflow orchestration system and techniques, as big data workflows consist of dynamic and heterogeneous analytical activities, where data arrives in different formats, at different volumes and at different speeds (Zhou and Garg, 2015).

Workflow mapping is to map the graph of analytical tasks to big data programming platforms (e.g. batch analytical task to Apache Hadoop, streaming analytical task to Apache

Storm), cloud resources, and edge resources. It also needs to consider different configuration possibilities (configuration of each big data programming framework, e.g., number of map and reduce tasks with Apache Hadoop in the context of batch processing, configuration of cloud resources, e.g., the type of resource and the location of datacenter; configuration of edge resources, e.g., type of edge device and network latency) This requires a cross-layer resources configuration selection technique in the big data orchestration system to select custom configurations from plenty of possibilities.

As a result, several challenges have emerged due to the complexity and dynamism of big data workflow including workflow specification languages, initialisation, parallelisation and scheduling, fault-tolerance and security. Since the heterogeneous and dynamic nature of cloud + edge resources bring additional challenges (we will discuss this at Cloud + Edge Datacenter level), these challenges further complicate those workflow-related challenges.

2.2.3 Big Data Programming Models/Frameworks Level

The processing of big data requires heterogeneous big data programming models, where each one of them provides a solution for one aspect. Within big data workflow, various computational models may be required for involved analytical tasks, where one analytical task may also need distinct computation models based on the characteristics of data (batch processing for static datasets, stream processing for dynamic datasets, hybrid processing for static and dynamic datasets). SQL and NoSQL models are also utilised for storing data to cope with volume and velocity of data. Therefore, understanding these models is essential in selecting the right big data processing framework for the type of data being processed and analysed.

These different models cover ingesting, storing and processing of big data. The MapReduce programming model (batch-oriented model) and stream programming model are used for data processing, NoSQL/SQL models are used for data storage, and message ingestion models are used for data importing. In this section, we will review these models and compare them to outline the main differences.

The complex and dynamic configuration requirements of big data workflow ecosystems calls for the need to design and develop new orchestration platforms and techniques aimed at managing: (1) sequence of analytical activities (formed workflow application) that needs to deal with static as well as dynamic datasets generated by various data sources; (2) heterogeneous big data programming models; and (3) heterogeneous cloud resources.

MapReduce Programming Model

The MapReduce programming model (Dean and Ghemawat, 2008) is a leading batch-oriented parallel data programming model that is intended for processing complex and massive volumes of data at once (static data) to gain insights. It was developed at Google Research, and relied on the following functions: Map and Reduce. The input data (finite large

datasets) is stored firstly in Hadoop Distributed File System (HDFS). Then, the input data is split into smaller chunks and then these chunks are processed in a parallel and distributed manner by Map tasks which generate intermediate key-value pairs. After that, these pairs are aggregated by Reduce function. Due to the finiteness property, this model has the capability to perform computation on data in stages, where it can wait until one stage of computation is done before beginning another stage of computation, allowing it to perform jobs just as sorting all intermediate results globally (Hirzel et al., 2013). Moreover, in respect of increasing future computation load, this model allows us to scale horizontally by adding more workers to cope with such loads. This model exploits data locality to schedule computation task to avoid unnecessary data transmission (Hu et al., 2014).

Stream Programming Model

In this model, data arrives in streams, which are assumed to be infinite and are being processed and analysed (in a parallel and distributed manner) as they arrive and as soon as possible to produce incremental results (Hu et al., 2014) (Hirzel et al., 2013). The sources of streams could be, for example, mobile and smart devices, sensors and social media. Thus, the stream computation in the stream programming model is assumed to process continuous incoming streams with low latency (i.e. seconds and minutes of delays), instead of processing a very large dataset in hours and more (Lin et al., 2016). There are two approaches to achieve this kind of processing/computation. The native stream processing approach processes every event as it arrives in succession, resulting in the lowest-possible latency, which is considered as the advantage of this approach; nevertheless, the disadvantage of this approach is that it is computationally expensive because it processes every incoming event. The micro-batch processing approach aims to decrease the cost of computation for the processing stream by treating the stream as a sequence of smaller data batches; in this approach, the incoming events are divided into batches by either time of arrival or once batch size hits a certain threshold, resulting in the reduction of processing computational cost, but could also bring together more latency (Keenan, 2016) (Lopez et al., 2016). With this model, stream computations are independent of each others, which means there is no dependency or relation among them. Moreover, in respect of increasing future computation load, this model allows us to scale vertically and horizontally to cope with such loads. Due to data-flow graphs implementing both data programming models, the stream-programming model can emulate batch processing. Apache Storm is one of the example of the stream processing platform. In addition to stream-oriented big data platforms, a number of stream-oriented services are offered by various cloud providers, which deliver stream-oriented big data platforms as services. Examples of these services are Microsoft Azure Stream Analytics and IBM Streaming Analytics.

NoSQL/SQL Models

For storing big data, there are two models, which are: NoSQL model and SQL model. The NoSQL models (MongoDB, Amazon Dynamo, Cassandra, HyperTable, BigTable, HBase) provide access capabilities reliant on transactional programming primitives in which a specific key allows a search for a specific value. The use of these access primitives results in improving the scalability and predictions of performance, making it suitable for storing huge amounts of unstructured data (such as mobile, communication and social media data). SQL data stores (Oracle, SQL Server, MySQL, PostGreSQL) organise and manage data in relational tables, where Structured Query Language as a generic language provides the ability to query as well as manipulate data. In essence, when transactional integrity (Atomicity, Consistency, Isolation, and Durability (ACID) properties) is a strict requirement, these data stores are more effective than NoSQL stores. However, both NoSQL and SQL data stores are likely to be used by future big data applications, and that is driven by data varieties and querying needs. SQL Models (Apache Hive, Apache Pig) provide the ability to query data over various cloud storage resources e.g. Amazon S3 and HDFS, based on structured query language. In respect of increasing future load, the NoSQL model allows us to scale horizontally using sharding or partitioning techniques to cope with this future load, while the SQL model has limited capability to cope with such loads.

Message Ingestion Models

The message ingestion model is a publish-subscribe messaging pattern that allows us to import data from various sources and inject it as messages (i.e. events/streams) into big data platforms for processing to produce analytical insights, where the senders of messages are called publishers and the receivers of messages are called subscribers. The stream computations are independent to each others, which means there is no dependency or relation among them. Moreover, in respect of increasing future computation load, this model can scale horizontally by adding more workers to cope with such load. Relying on these models, message ingestion systems (such as Amazon Kinesis, Apache Kafka) acheives a durable, high-throughput, fault-tolerant and low-latency queuing of streaming data.

Amazon Web Services (AWS) Kinesis is a cloud based stream platform offered by Amazon. It provides powerful services that allow working easily with real-time streaming data (to load and analyse continuous data) in the AWS cloud, and the ability to develop and build custom streaming data applications to meet specific needs.

Hybrid Models

To support applications requiring both batch and stream data processing, hybrid models are developed. An example of a cloud service that implements hybrid data programming models (batch and stream) is Google cloud Dataflow. It is a Google fully-managed service for stream data processing and batch data processing as well. Dataflow is an unified execu-

tion framework and programming model for creating and executing both batch and stream pipelines to load, process and analyse data, without having to pay attention to operational tasks such as resource management and performance optimisation. As an execution framework, it handles the lifetime of resources transparently and provisions resources on demand to reduce latency while at the same time maintaining high utilisation efficiency. Moreover, as a unified programming model, it uses the Apache Beam model that eliminates programming model switching cost between batch and streaming mode by providing the ability for developers to represent the requirements of computation without taking into consideration the data source.

Lambda Model − A batch-first approach uses batching for unifying batch and stream processing, where data streams are treated as micro-batches (collection of small batches). It supports batch processing for historical datasets to get insights according to the needs of users, and stream processing via micro-batching approach, which is suitable for applications where the data collection and availability through dashboards have time delays, and such data needs to be processed as it arrives (Kiran et al., 2015). Lambda model comprises three layers. Batch layer as a first layer is responsible for storing the master dataset and periodically precomputing the views of batch data. Speed layer as a second layer is responsible for processing online data as it is received in near real-time fashion to minimise latency. Serving layer as a third layer is responsible for consolidating both by combining the results from batch and speed layers to answer user queries. Lambda architecture achieves two properties of big data, which are velocity and volume. By using such architecture, users can determine which data parts need stream or batch processing in order to improve their data processing costs.

Kappa Model − A stream-first approach that considers all data as streams, whether such data is batch data or stream data. In contrast to Lambda architecture, this architecture, in favour of simplicity, dispenses the batch layer. Thus, there is no periodical recomputation for all data in the batch layer, instead the Kappa architecture performs all data computation in one system (i.e. stream processing system) and executes recomputation only when there is a change in business logic by rerunning historical data. This accomplishes by utilising a powerful stream processor that is able to handle data at a higher rate than incoming data rate as well as a scalable streaming system for data retention. Kappa architecture comprises two layers. The speed layer manges processing of stream data, while the serving layer is responsible for answering user queries, similar to the serving layer in the Lambda architecture. Apache Spark is an example of such big data processing platform that combined more than one programming model.

Comparison of Properties of Big Data Models

The comparison between big data models including a batch programming model, stream programming model, NoSQL/SQL models and message ingestion models is given in Table

2.1. This comparison is based on five properties, which are data flow (a pattern in data computation implementation), data volume (the size of data), relation (the relationship between the computation implementation of functions), scalability (the capability of increasing resource(s) capacity in response to the future load), and plus and negative points.

As there are different big data models, several big data platforms and services have been developed such as Apache Hadoop, Spark, Storm, Flink, Amazon Kinesis, Azure Stream Analytics, Google Cloud Dataproc and IBM Streaming Analytics. (Rao et al., 2018) provided a survey of various big data systems.

2.2.4 Cloud and Edge Datacenters Level

The cloud and edge infrastructures that provide heterogeneous and distributed compute and storage resources are viable solutions for executing and managing big data workflows, and fulfilling the SLA and QoS requirements defined by users. However, the process of executing and managing such types of workflow in cloud + edge datacenters is a complex resource and data orchestration task. The complexity comes from the composite data flow pattern, various computational models involved in the data pipeline, various big data programming frameworks needed for those computational models and different types of cloud and edge resources required during the workflow orchestration. The heterogeneous and dynamic nature of cloud + edge resources bring additional challenges (selection of optimal resource types and their configurations, resource failures and so on), where these challenges also further complicate the workflow-related and data-related challenges, and therefore present a unique cross-layer challenge. The key issue at this level is the real time selection of the optimal configurations of cloud and edge infrastructures for given heterogeneous workflow components taking into consideration SLA and QoS requirements defined by workflow users based on the context of application. This includes the following challenges: cloud platform integration and cloud + edge resources management.

In summary, for managing and executing the big data workflow application, several requirements need to be considered due to complex interaction of the three layers i.e. (1) big data workflow, (2) the different big data models and different big data analysis applications (such as batch processing, stream processing, SQL, NoSQL, Ingestion), and (3) cloud + edge computing environments. In the next section, we will identify these requirements.

2.3 Requirements of Big Data Workflow in the Cloud

Based on the extensive literature review and study of the characteristics of big data workflow applications, we discuss the key requirements for their orchestration over heterogeneous cloud resources (CPU, Storage, and Software Defined Networking SDN Infrastructure). The heterogeneity at the workflow level (different analytical activities deal with real-time and historical datasets), big data programming model level (batch, stream or hybrid processing),

Table 2.1: Comparison between big data programming models

Property	**MapReduce Programming Model**	**Stream Programming Model**		**NoSQL/SQL Model**	**Message Ingestion Model**
		Native	**Micro-batch**		
Data Flow	Static	Streaming		Transactional	Streaming
Data Volume	Known (finite large datasets)	Unknown (infinite continuous events – small data)	Unknown (infinite continuous events – micro-batches (a batch is finite set of streamed data))	Known (structured data)	Unknown (infinite continuous events – small data)
Relation	Dependent and synchronous computations	Independent asynchronous computations	Bulk synchronous computations	-	Independent and asynchronous computation
Scalability	Horizontal scalability (adding more workers)	Vertical and horizontal scalability (increasing the capacity of workers as well as adding more workers)		NoSQL: Horizontal scalability (using sharding or partitioning technique) SQL: limited scalability (manual)	Horizontal scalability (adding more workers)

continued . . .

...continued

Property	MapReduce Programming Model	Stream Programming Model		NoSQL/SQL Model	Message Ingestion Model
		Native	Micro-batch		
Pros (+)	+Extensive and distributed data processing for static data +No need for ingestion system +Estimation of completion time of data processing task	Both: +Extensive and distributed processing for real-time and near real-time data +Store data portion in memory or no store +Low latency (milliseconds for native and seconds for micro-batch model (Venkataraman et al., 2017)) Native (Venkataraman et al., 2017): +No barrier and thus no centralised communication overhead +Low latency during normal execution Micro-batch (Venkataraman et al., 2017): +Efficient fault-recovery and scaling due to the use of barriers		SQL: +Multi-row ACID properties +Relational features (e.g. join operations) NoSQL: +Extensive and distributed data processing support with limited flexibility (Cai et al., 2017) +Support various data types and data speeds +Update schema on the fly	+Extensive and distributed processing for real-time and near real-time data +Different message processing semantics (at-least-once, exactly-once and at-most-once)

continued ...

...continued

Property	MapReduce Programming Model	Stream Programming Model		NoSQL/SQL Model	Message Ingestion Model
		Native	Micro-batch		
Cons (-)	-All data need to be stored in storage system -Redundant and excessive processing -High communication cost -High latency	Both: -Need ingestion system -High overhead Native (Venkataraman et al., 2017): -High overheads during adoption	Micro-batch (Venkataraman et al., 2017): -Need blocking barrier following every batch - Communication overheads	SQL: -No strong support for extensive and distributed data processing (Cai et al., 2017) -Offline database to update schema NoSQL: -No relational features (e.g. join operations)	-Limit of message size -Overhead -Balancing the data that coming from various data sources

and cloud-level (cloud datacenters and edge resources) leads to diverse requirements that are described as follows:

1. *Compute/CPU Resources Provisioning Requirement* − To execute tasks/analytic activities related to a big data workflow, diverse mix and type of compute/CPU resources (e.g. virtual machines, lightweight containers) are required. These resources are provisioned in static or dynamic way (Rodriguez and Buyya, 2017) according to the need of such workflow task/activity and type of underlying big data programming model used (e.g. batch processing, stream processing or hybrid). Provisioning the necessary compute resources for executing big data workflow is not the end of story, monitoring and managing those resources in a dynamic execution environment is also needed because those resources are provisioned and released on demand due to changes in data volume and velocity and resource-level failures (Kashlev and Lu, 2014).

2. *Storage Requirement* − By taking the decision to move and execute big data workflow using cloud infrastructure, the next decision that will be taken implicitly is moving and storing big data products of such application in the cloud. Thus, we need to intelligently provision the cloud storage to store data and feed the data to different big data programming models at different stages of the workflow execution including, for example, choosing the right cloud storage resource, data location (hence requires novel indexing and metadata management techniques) and format.

3. *Data Movement Requirement* − For data residing out of the cloud, such data needs to be transferred to the cloud and stored before being processed by big data workflow. In addition, the stored datasets may reside across different locations and these locations may differ based on geographical deployment of cloud datacenters where compute and storage resources are hosted, so dynamically transferring these large datasets to between compute and storage resources presents new research requirements such as bandwidth allocation and data transfer latency and throughput management. For example, transferring a large amount of data (i.e. large datasets) needs a high bandwidth. In addition to an external network (i.e. Internet), dealing with internal networks of the cloud (networks inside the cloud itself) is also needed. The performance of such networks as is not the only thing required, but dealing with its structure and configuration is also needed. One interesting area of research that will emerge includes how to exploit SDN-based infrastructure within clouds to create more dynamic and flexible data movement techniques and protocols driven by SLA and QoS needs of workflows.

4. *Synchronisation and Asynchronisation Requirement* − In big data workflow, there may exist control and data flow dependencies across analytics tasks. For the dependent tasks, the run-time synchronisation is required at both data flow as well control flow levels. Moreover, the execution of dependent tasks requires dynamic synchronisation of the states (e.g. output of upstream tasks forms the basis of input data to one or more downstream

tasks) of upstream and downstream analytic tasks. On the other hand, for independent tasks, no such run-time state (data plus control flow) synchronisation requirement exists. In summary, the data and control flow requirement is one of the most important workflow choreography requirements to be considered because it directly impacts the correctness of workflow execution and end-to-end performance, to say the least.

5. *Analytic Task Scheduling and Execution Requirement* − Provisioning the necessary virtual resources for big data workflow is not the end of the story to running such workflow, scheduling and coordinating the execution of workflow tasks across diverse sets of big data programming models (Ranjan et al., 2017) as well as balancing compute resource utilisation across the tasks also being required (Zhao et al., 2014). In addition, partitioning big data workflow into fragments and parallelising the execution of those fragments using parallelism techniques is important for the scheduling process, which allows it to schedule the partitioned workflow fragments separately on different compute resources to maximise performance and reduce the complexity of scheduling. Moreover, during the execution of a task, the input data for this task is moved to compute resource, the output data is generated and in general, data products' provenance is produced (Liu et al., 2015b). Therefore, tracking and capturing provenance of data is also needed.

6. *Service Level Agreement Requirement* − The execution of big data workflow may need to meet quality attribute measures defined by users via SLA. These measures, requirements of QoS, are stated in SLA in order to ensure reliable QoS (Beloglazov et al., 2012). For example, one quality might be execution deadline, which means the execution of workflow should be completed with strict time constraint (i.e. on or before deadline). Therefore, we need not only to execute big data workflow in the cloud, but also meet user-defined QoS requirements.

7. *Security Requirement* − Moving big data computation along with the associated datasets to the cloud imposes the need to secure both data and computation. This introduces a number of challenges that require solutions that go well beyond standard encryption approaches, but include challenges such as private (anonymous) computation, verification of outcomes in a multi-party setting (Dong et al., 2017), placement of components according to security policies (Mace et al., 2011), etc. Thus, applying security protection to workflow tasks during their execution and to the data itself provides a high level of security when running such workflow in the cloud.

8. *Monitoring and Failure-Tolerance Requirement* − Big data workflow comprised of data-intensive tasks and the execution of those tasks is usually a lengthy process. Therefore, monitoring the execution of workflow is needed to ensure that everything is streamlined and executed as anticipated. Moreover, failures could happen at any time during the workflow execution, so that handling those failures when they occur or predicting them before they happen is also needed.

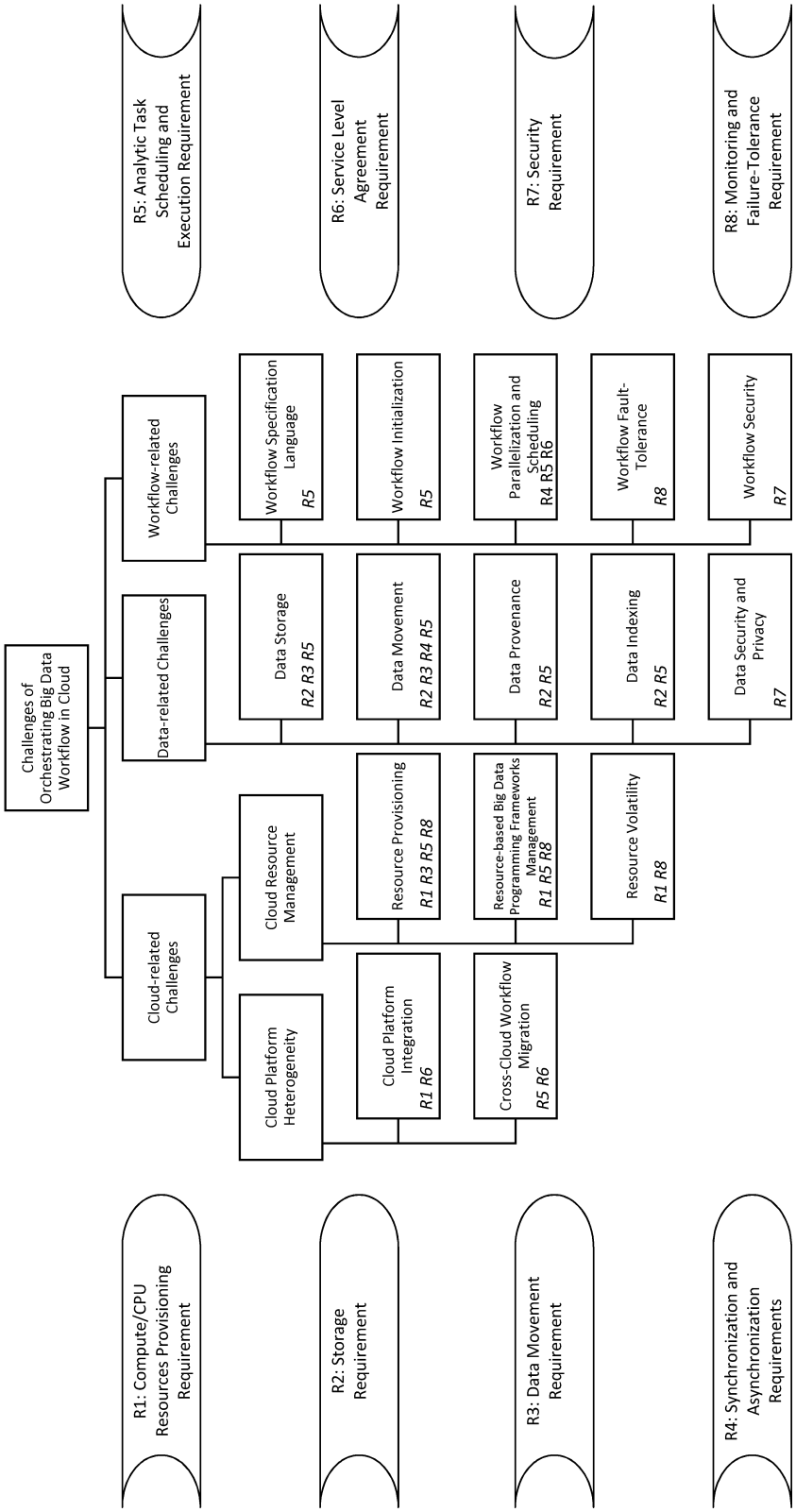


Figure 2.2: A taxonomy of challenges for orchestrating big data workflow in the cloud with the mapping of aforementioned big data workflow requirements to these challenges

2.4 Research Taxonomy for Orchestrating Big Data Workflow Applications

The complexity and dynamic configuration requirements of big data workflow ecosystems calls for the need to design and develop new orchestration platforms and techniques aimed at managing: (1) sequence of analytical activities (formed workflow application) that needs to deal with static as well as dynamic datasets generated by various data sources; (2) heterogeneous big data programming models; and (3) heterogeneous cloud resources (Ranjan et al., 2017). The orchestration process contains a set of programming tasks, which are workflow composition, workflow mapping (to map the graph of analytical activities to big data programming platforms and cloud/edge resources), workflow QoS monitoring (to oversee QoS and SLA statistics at runtime for each activity in this graph such as alert delay, load, throughput, utilisation) and workflow dynamic reconfiguration (to reconfigure workflows in composite computing infrastructure comprised of cloud, edge and multiple big data platforms), all for guaranteeing consistency and adaptive management (Ranjan et al., 2017). The requirements posits numerous challenges that do not occur when executing those workflows in conventional computing infrastructure. This section outlines and discusses the research challenges (cloud-related, data-related and workflow-related challenges) and associated taxonomy with the mapping of big data workflow requirements in the cloud discussed in previous section to these challenges (Figure 2.2).

2.4.1 Cloud-related Challenges

The cloud related challenges can be viewed from four dimensions: Cloud Platform Heterogeneity, Resource Management, Data Management and Storage, and Data Security and Privacy.

Cloud Platform Heterogeneity

The cloud platforms offered by different vendors are heterogeneous and varies in their capabilities. Following details challenges associated with this dimension:

1. *Cloud Platform Integration:* Before provisioning cloud resources, the mapping of big data programming models (that realise different workflow activities) to cloud platforms is required. Each cloud provider defines a specific set of API for supporting such mapping and deployment. This means that the application programming process varies across different cloud providers and for each one of them, the user should learn how to interact with different cloud providers that support heterogeneous APIs (Kashlev and Lu, 2014). Thus, connecting to multiple cloud platforms is more complex since the workflow application programmer and/or administrator needs to know the specific programming primitive and patterns relevant to APIs of underlying cloud providers. Accordingly, the user needs to learn several vendor-specific virtualisation formats, pricing policies and other

hardware/software configurations, yielding much complex integration challenge. Overall, dealing with integration challenge is complex and requires novel tools, techniques, and API stack for simplifying the mapping and deployment of complex big data workflows to heterogeneous cloud providers.

2. *Cross-Cloud Workflow Migration:* After mapping and deploying activities of a big data workflow in one cloud, migrating such workflow activities with large datasets to another cloud is a non-trivial process. The users and/or administrators could need to migrate their workflows from one cloud to another because, for example, they might aspire to specific QoS features in the target cloud or better price (Kashlev and Lu, 2014). In the target cloud, different types of heterogeneous resources (e.g. virtual machines, storage types, network types) are there and selecting the right number and configurations of resources is a crucial (i.e. remapping and re-deployment) step (Kashlev and Lu, 2014). Further migrating (mapping + redeploying) workflow activities to other clouds also means moving large datasets and data platforms, which may be a costly and time-consuming task. As a result, the integration challenge in a cloud and/or across multiple clouds (i.e. difficulties of providing a uniform and transparent way to access to different clouds and provision virtual resources from different clouds) is complicated in big data workflows orchestration (Ranjan et al., 2017) (Kashlev and Lu, 2014).

Cloud Resource Management

Big data workflow execution in the cloud requires the appropriate selection of cloud resources and their configurations including the provisioning such virtual resources on demand, and creating and managing those resources as well as coping with the dynamic nature of cloud resources.

Resource Provisioning As the execution of big data workflow will be carried out in the cloud, the first and important step is selecting the right configuration of virtual resources (virtual machine and/or virtual CPU, storage, and network), which is a challenging decision in case of considering various types of resources offered by various vendors and becomes even harder when considering different instances from different clouds to achieve the intended goal. Furthermore, when the selection of edge resources come into the picture, new challenges are being added that include the consideration of diverse edge devices, their hardware features and virtualisation support with container technologies, and the conflict SLA and QoS requirements (Ranjan et al., 2017). In addition, the resource selection decision should meet the degree of parallelism needed for data processing tasks composed in a workflow. For example, considering particular configuration of a cloud such as Google Compute Engine with 18 predefined instance types, it is difficult to find an optimal configuration in order to achieve an optimal execution time, as the configuration selection problem is generally an NP-complete problem (Ranjan et al., 2017) (Kashlev and Lu, 2014). Thereby, with differ-

ent stages of resource selection, scheduling workflow activities on the selected resources at each stage to run them is also an extremely hard problem (workflow scheduling problem). Also, when considering various resource configurations provided by multiple cloud providers, comparing those configurations to find the best one for a given set of workflow activities is an open research problem as we note in (Ranjan et al., 2015) (Ranjan et al., 2017); it is not only for workflow activities, but also involving implicitly big data programming frameworks, a cross-layer problem (at Infrastructure as a Service (IaaS) and Platform as a Service (PaaS)-levels) (Ranjan et al., 2015). In other words, the resource configuration search space grows exponentially when we consider each analytical task composing the workflow.

Big data workflows involve various big data workloads, and these workloads have different resource requirements. For batch workloads, the storage requirements dominate, and for streaming workloads, the communication requirements dominate, while for transactional workloads, the computational requirements dominate (Ranjan et al., 2015). Considering different types of workloads in complex workflow scenarios require configuration selection mechanisms to have intelligence in order to assist them in reducing resource contention that can occur due to the interference of workload. This will require determination of those workloads (aka virtual machines) that can be combined in a same physical environment. Obtaining resource contention information needs both offline benchmarking and real time SLA monitoring mechanisms.

After the configuration selection, the next step is just to call the cloud provider specific API which will instantiate the resources we need for example virtual machine/CPU instance, storage space, network IPs and network bandwidth (in case of cloud that support software defined networking). Such a process is not as easy as it seems at first glance because various aspects need to be taken into consideration such as resource location. Big data workflows include multiple data analysis tasks and those tasks are executed in several stages, where each stage might require specific cloud resources. Those resources can be configured differently in order to achieve the intended requirements, but the level of granularity and flexibility is hard to determine (Zhao et al., 2015c).

As a result, the problem of resource configuration selection exists across various types of cloud resources since the need here is to allocate cloud resources (for example virtual machine, storage, network IP, network bandwidth, etc.) to workflow activities and underlying big data programming frameworks. Thus, the allocation of cloud resources at IaaS-level to big data programming frameworks at PaaS-level is not any more a conventional resource maximisation or even time minimisation problem, however it includes simultaneous objectives and configuration dependencies over various IaaS-level resource and big data programming platforms (Ranjan et al., 2015).

Resource-based Big Data Programming Frameworks Management Orchestrating heterogeneous workflow tasks over the cloud requires cloud resources (e.g. virtual CPU, storage and network) as well as big data programming frameworks (for example Apache Ha-

doop, Apache Spark, NoSQL). Therefore, the management of PaaS-level big data programming frameworks (that implement various software-based data processing primitives (such as batch processing or stream processing) on IaaS-level resources (that provide computing capacity to those frameworks) is needed in the context of big data workflows. Achieving such demand is a complex challenge as it requires determining the optimal approach to automatically select the configurations for both IaaS-level resource and PaaS-level framework to consistently accomplish the anticipated workflow-level SLA requirements, while maximising the utilisation of cloud datacenter resources (Ranjan et al., 2015).

Resource Volatility As mentioned earlier, the loss of the provisioned resources often happens due to different failures (Kashlev and Lu, 2014). As well, big data workflows consist of complex big data tasks/analytic activities, and thus the loss of state of analytical processes executed by the big data programming framework could happen at any time. Accordingly, several challenges have emerged as a consequence of the complexity and dynamic nature of cloud resources and big data workflows (i.e. different platform and infrastructure requirements for each workflow task, and dynamic processing requirements of each workflow task which are determined by either data flow or control flow dependencies).

During the execution of workflow tasks involved in workflow application, we consider the task completed when the following steps are executed successfully before the provisioned virtual resources being are terminated and released: (1) data computation and processing is done and (2) the output data as a result of this computation and processing is stored in temporary or permanent storage. However, the user may at any time and under any circumstances terminate the virtual resource while the execution of a workflow task is still under way or the aforementioned steps are not yet completed. This highlights a challenge to deal with failures of virtual resources that originated not from the resources themselves but from user actions. Even after a successful completion of task executions, storing output data products produced as a result of the execution of a big data workflow application is a challenging task, since those data products are big data products and the user in most cases tries to avoid paying for unused virtual resources after completion of execution by terminating and releasing those resources immediately (Kashlev and Lu, 2014). Moreover, the user might need to add new analytic tools and libraries to virtual machines to be used later on. Those products could be lost in the case of terminating VM if precautionary actions are not taken. Furthermore, workflow may rely on specific libraries and packages to run, where different tasks might have different dependencies. The volatile nature of cloud resources means that configuring a virtual machine with the required dependencies is actually not a one-time procedure, where such configuration will be lost in the cases of the VM being terminated.

In addition, the problem of resource volatility becomes more complicated in big data workflows when considering the volatility of resource at different levels (VM-level, big data progressing framework-level and workflow tasks-level). The resource volatility at VM level is the sense of losing the state of the virtual machine in terms of data stored in Random

Access Memory (RAM) and/or non-persistent storage. At big data programming framework level (such as Apache Hadoop), it is the sense of losing the state of mapper and reducer processes which we cannot capture at VM level; while at workflow tasks level it includes the loss of analytic computation completed so for, which may incur additional cost or delay in execution with the best case. Overall, dealing with resource volatility in the context of big data workflows is more complex task.

2.4.2 Data-related Challenges

Moving the execution of big data workflow to the cloud means dealing with end-to-end orchestration operations relevant to securing and managing data including storage, movement, provenance, indexing, security and privacy. Each of these orchestration operations individually is a very challenging task (Zhao et al., 2015c) and to large extent this remains a fertile area of research in the context of big data workflows.

Data Storage

As big data workflows support the composition of heterogeneous data processing tasks into data pipelines, the different types of data flow (batches of data or streams of data) associated with different big data programming models that form part of workflows exist. For instance with message queueing (such as Apache Kafka) and stream processing (e.g. Apache Storm), the streams of data flow into Kafka via Kafka producer or into Storm cluster via spout, while with batch processing (e.g. Apache Hadoop), large datasets should be stored over cloud storage and then fed into Hadoop cluster via HDFS. Since the execution of big data workflows will be carried out in the cloud, the different storage needs for heterogeneous workflow tasks lead to different challenges in dealing with cloud storage resources to satisfy their needs.

With batch processing tasks, these tasks communicate using files (i.e. large files). The one or more output files (output datasets) generated by each analytical task become input datasets to other analytical tasks, and those datasets are passed between tasks using data movement techniques or through shared storage systems (Cafaro and Aloisio, 2011). Thus, the large input datasets stored out of cloud must to be moved to and stored in the cloud before analytics can start, and the intermediate output datasets generated during the processing as well as the final large output datasets produced upon the completion of processing are required to be put in storage in the cloud, where the data can be thrown out after analytics is done. On the other hand for stream processing tasks, the analytic result and some input data for provenance can be stored. Accordingly, different storage needs for workflow tasks incur different computing "network and storage" costs, where dealing with that is so complicated as compared with traditional application workflows. Also, choosing which cloud storage resources to use to store data for batch and stream processing tasks has a direct implication on the incurred computing cost. Such selection is a challenging task and becomes more

difficult when taking into consideration where the data will be residing, which data format will be used and where processing will be done.

Data Movement

As the execution of big data workflows will be carried out in the cloud, transferring data to the cloud, within the cloud and/or between clouds is needed to proceed in execution. The different data flows associated with different big data programming models poses different demands for data movement, so that the problem of data movement is more complicated in the context of big data workflows.

For batch processing tasks, transferring input datasets stored in local machine to cloud storage or data in bulk through hard-disks is required before those tasks are started. Similarly, intermediate datasets produced by those tasks must be moved among execution virtual resources and the outputs resulting from their execution must also be transferred to the following tasks or to cloud storage. Thus, coping with the movement of high volume of historical data to the cloud, and between clouds for batch processing tasks is a non-trivial challenge, because this movement is a costly and time-consuming process as well as having direct implications (in term of expensive execution overhead). Moreover, avoiding both the suspension of some workflow tasks to perform data movements (Tudoran et al., 2016) and the waiting time until data is moved to the execution environment are important issues that should be addressed with this challenge.

On the other hand for stream processing, there is no bulk data to be transferred as data is continuously coming from data stream sources and ingesting into data pipelines, however the streams of data generated by data producers should be moved to the cloud resources where stream processing tasks are executed, and that is incurring data transfer time as well as data transfer cost in case of transferring data between clouds, which are relatively small compared with moving high-volume batch processing of historical data. The challenge here is avoiding or at least minimising the delay in transferring real-time data as the freshness of this data so important, as well as analytical results for streaming data.

Accordingly, different data flows affect data movement latency (high with batch processing due to moving vast volumes of historical data and low for stream processing as the size of the stream is small), as well as incurring different network costs (high with batch processing and low with stream processing). Hence, the complex problem in data movement is no longer just in moving one type of data flow, but heterogeneous workflow tasks communicate using different types of data flow where each one has its implications on the movement of data in big data workflows. In addition, despite the different types of data flow in data pipeline, transferring the large-scale application data that have been collected and stored across geo-distributed datacenters may be subject to certain constraints (e.g. the data size, the network burden or the General Data Protection Regulation (GDPR))), which determine which of these data can be migrated and which cannot (Hung et al., 2015) (Convolbo et al.,

2018) (Chen et al., 2018b). Thus, the problem of data movement becomes even harder if taking into consideration such data movement constraints and more complex if taking into account several constraints together (such as data size and network burden) when moving data across geo-distributed datacenters during data processing time.

Data Provenance

The data provenance describes the origins and historical derivations of data by means of recording transformation provenance (those transformations that are in charge of creating a certain data element) and data provenance (derivation of a given data element as of which data elements) (Glavic, 2014). An instance of such transformation is big data workflow that generated data product provenance. Dealing with provenance for such workflow is a challenge due to the properties of big data (i.e. 3Vs of big data), the characteristics of the execution environment (highly-distributed and dynamic), the distributed provenance information of data elements, the cost of transferring such information together with data elements (large amount of information) and the complexity of evaluating queries over such information (Glavic, 2014).

The track of provenance of historical data deals with a large volume of finite datasets, so that the provenance collection cost/overhead is high and the collecting provenance can be grown larger than the size of data being described. For that, provenance data of historical data is too large, and the storage requirement is becoming an additional challenge. Tracking of provenance of streaming data deals with infinite streams, non-deterministic behavior (e.g. high input rates, delay), stream aggregation (combining multiple streams into one by streaming workloads), ordered sequences (order of streams) and performance requirements (e.g. provenance generation and retrieval) (Glavic et al., 2011), makes it a hard challenge. As well, the fine-grained provenance data generating from small datasets (e.g. streaming data) can be large in size, so that the storage requirements (Huq et al., 2011) and provenance collection overhead are the associated challenge, but the communication overhead is the dominant challenge. For that, this hard challenge becomes even greater since the demand is to trade off expressiveness (provenance representation) with a moderate overhead during provenance processing and retrieval.

Moreover, the dynamic and distributed execution environment introduces the demand for capturing and querying distributed provenance of data products, which makes the audit, track and query of distributed provenance more complex (Malik et al., 2010). In addition, distributed transformations of a data item incurs collecting both the provenance of such an item that refers to data and transformations out of each virtual machine that was in use when creating such an item, where the transfer of data items with their provenance means to transfer large amount of information among virtual resources. That is leading to the need for distributed query solution for big provenance (Glavic, 2014). As a result, the problem of data provenance is complex, but its importance lies in the fact that tracking of data

provenance allows understanding and reusing workflows (Zhao et al., 2015c).

Data Indexing

It is data structure that aims at creating indexes on datasets in order to accelerate data access as well as data query operations. Data indexing is an issues in big data workflows because (1) each workflow step needs different datasets based on nature of analytic computation, (2) datasets are tremendous, highly dimensional, heterogamous and complex in stricture (Chen et al., 2013), and (3) execution environment (i.e. cloud) is distributed, where all of that complicates the developing of the indexing approach. Moreover, challenges exist in knowing the type of data being indexed and the data structure being used, keeping the cost of creation and the cost of storage space (for storing indexes) low or moderate, and specifying index generation and size.

Data Security and Privacy

In cloud and edge computing, data security remains a major concern. When the execution of big data workflow is performed on cloud/edge resources, big data being processed by workflow tasks will be stored in and accessed from the cloud/edge. Thus, security aspects of workflow data include cryptography (to secure actual data (Mattsson, 2016)), integrity (to ensure data consistency and accuracy), access control (to enforce restrictions on access to data), authentication (to verify the identity of an individual with or without disclosing such identity (He et al., 2018) (He et al., 2016)), data masking (to replace sensitive data with masked data), tokenisation (to replace original data with random value of the same data format (Mattsson, 2016)) and privacy (to restrict the collection, sharing and use of data). Since investigation of these aspects is a large topic in itself and is beyond the scope of this chapter, we list some of security issues and briefly review them along with the related approaches in Appendix A.4.

2.4.3 Workflow-related Challenges

Workflow Specification Language

Workflow specification language defines workflow structure and its task. For big data, there are different big data models available such as batch processing (MapReduce), stream processing, SQL, NoSQL, where each one of them have their own way to specify computation, so that further functionality and flexibility in the specification of workflow is required to support those models. Consequently, the hard challenge here is to create a workflow-level specification language that can be more human-friendly and can be automatically transformed to programming model specific specification language (e.g. MapReduce in the context of Apache Hadoop, continuous query operators in the context of Apache Storm, relational queries in the context of SQL, non-relational queries in NoSQL databases). Moreover, this challenge

becomes more complicated if the specifications of cloud resource and big data management are taken into consideration. The former challenge is in specifying the cloud resource specification as part of the workflow specification language at least at high level and that could be in terms of QoS, performance, security/privacy constraints, and the latter challenge is in specifying big data management specification also as part of specification language at least at high level and that could be in term of data format, storage constraints, data movement restrictions. In addition, for Multicloud architecture, a standard specification language for big data workflows is needed to make such workflows portable and scalable across multiple clouds.

Workflow Initialisation

Workflow initialisation aims to divide a workflow into several small parts called workflow fragments (or fragments for short) to allow parallel and distributed processing, where each fragment contains part of the workflow activities and their data dependencies (Liu et al., 2015b) (Liu et al., 2014). It could be a constraint-based process to take into account some constraints such as compute resources or minimising data transfer during partitioning a workflow. Since big data workflows include multiple data analysis tasks and those tasks are executed over virtual resources provisioned from one or more clouds in a parallel and distributed manner, such an initialisation process is needed. In other words, workflow initialisation is needed for executing big data workflows in the cloud in a parallel and distributed manner.

Workflow initialisation is a non-trivial task since it necessitates to take into account the task and data dependencies within the workflow, and to avoid cross dependency. It becomes harder if considering the aspects of data management (storage and indexing). For the data storage aspect, we need to consider the different needs of storage resources for heterogeneous workflow tasks during the partitioning process, so that fragments of workflow respect these needs. For the data indexing aspect, we need to consider the index data for datasets during the partitioning process since each workflow step requires various datasets based on the nature of analytic computation, so that the data required for these workflow steps can be searched and retrieved quickly.

Furthermore, this challenge becomes more complicated if other restrictions are taken into account. For example, multisite execution, data transfer, storage resource constraints or balancing the activities of workflow in each workflow fragment whilst lessening the linked edges amongst various workflow fragments (Liu et al., 2015b).

Workflow Parallelisation and Scheduling

After initialising the workflow, the partitioned workflow fragments are parallelised and scheduled on cloud resources in order to be executed on those resources. For workflow parallelisation, various techniques are utilised in order to produce concrete executable workflow tasks for the execution plan (Liu et al., 2015b). The workflow parallelisation results included in

the workflow execution plan is a decision of parallelising workflow tasks to execute them in parallel. For big data workflows, they are parallelisable across all two levels: big data programming framework level and workflow activity level. At big data programming framework level, the frameworks (e.g. Apache Hadoop, Apache Storm) are workflows themselves, for example, workflow of Map/Reduce tasks in Apache Hadoop, workflow of spout and bolts tasks in Apache Storm. While at workflow activity level, each activity is heterogeneous and mapped to a different machine.

The workflow scheduling needs to cater for above super-workflows and then find an optimal resource allocation plan for lower level cloud resources, which is an extremely hard research problem. This complexity comes from several aspects that have to be taken into consideration to create an efficient scheduling plan. This plan is aimed at balancing resource utilisation across sub-workflows involved in big data workflow and making the execution complete to achieve the desired objectives. It will also be revised in response to unexpected changes occur at runtime such as changes in data velocity and resource availability. The heterogeneity of data analysis tasks involved in big data workflows complicates the situation. Considering the location of data during the task scheduling period is important (Liu et al., 2015b). In other words, task scheduling has to be aware of the location of data to minimise data movement. Moreover, the user quality of service requirements need to be considered. Furthermore, the use of Multicloud architecture is a complex aspect since it necessitates to be aware of the arrangement of resources and big data programming frameworks in this architecture in order to map the fragmented workflow parts or tasks to available workers in addition to utilise resources in this architecture by considering data location during task scheduling.

Furthermore, when workflow scheduling utilises edge resources, new challenges come into the picture to efficiently map and deploy data analysis tasks on resource constrained edge devices including the achievement of three core properties of a container, which are isolation, orchestration and scheduling (Rao et al., 2018). The lightweight hardware with lightweight containerisation software is needed (von Leon et al., 2019). Finding the optimal resource selection and allocation plan for executing data analysis tasks at edge resources should take into account the characteristics and hardware constraints of edge devices, and the heterogeneity of these tasks, where this plan could be part of a full allocation plan for cloud and edge resources. The other challenge is adapting heterogeneous big data workloads (i.e. data analysis tasks) from VM-based workload to container-based workloads for container platform, Kubernetes. This also includes the challenge of creating efficient container images for those workloads. Moreover, managing and monitoring the containers for big data workloads over edge resources with Kubernetes is complicated due to the dynamic nature of edge resources and their changing performance, and the need to define runtime configuration and deploy them into the container environment. This container management process becomes even harder with the need to maintain SLA and QoS requirements on those constrained resources

(such as execution cots and data processing throughputs), and to respond to unexpected changes at runtime (Ranjan et al., 2017). Overall, the complexity of orchestration is increased when scheduling is considered in an edge environment virtualised using lightweight containers.

Finding optimal virtual resources to allocate them to workflow tasks and underlying big data programming frameworks, and the optimal configurations for both IaaS-level resource and PaaS-level big data programming frameworks helps to achieve SLA scheduling.

Workflow Fault-Tolerance

Workflow fault-tolerance intends to handle failures that occur during the execution of workflow and assure its availability and reliability (Liu et al., 2015b). Since big data workflow is complex, dynamic, and heterogeneous (the complexity and dynamism of big data ecosystems), and its execution is usually a lengthy process and will be carried out in a dynamic environment, the failures may happen at any time for numerous reasons such as the change in execution environment, the loss of compute resource, error during analytical task exaction or unhandled errors during task execution. Thus, failure management in workflows is much more complicated as things can go wrong at any level (workflow-level, big data processing-level and cloud-level). It becomes harder and harder with big data workflow consisting of data- and compute-intensive tasks along with the need to predict failures and accordingly take appropriate actions in order to avoid additional expensive costs that could be incurred if failures occur such as re-computation and re-transferring data costs. As a result, developing a holistic failure-management/tolerance technique is a challenging process, and most likely ends with a suite of failure management algorithms specific to each workflow activity type.

Workflow Security

In the context of big data workflow, securing big data (Cuzzocrea, 2014) is not the whole story, it is considered as a part in preserving workflow security. The other part is guaranteeing the security of workflow logic and computation. As big data workflow is data-centric, ensuring the security and integrity of processing or computation carried out on big data in addition to data security are the main challenges. The lack of interoperability is a particular issue since the underlying execution environment used for running such workflow is distributed and heterogeneous by nature (Kashlev and Lu, 2014). Moreover, with the difficulties of managing authentication and authorisation in such workflows, preserving such levels of security becomes even harder and more challenging. The heterogeneous data processing involved in workflow may require different security needs adding more complexity and makes ensuring security at workflow level a complex task (Mace et al., 2011).

2.5 Current Approaches and Techniques

Presenting several orchestrating big data workflow in the cloud is important, but knowing how to resolve them is crucial. This section reviews the current approaches and techniques related to the presented research taxonomy.

2.5.1 Cloud Platform Integration

The aim of cloud platform integration is to mask the heterogeneity among different cloud platforms offered by various cloud providers in order to provide a uniform way to access clouds of various vendors as well as to provision, consume and manage resources from different clouds. There are two following generic approaches to resolve the incompatibilities between different cloud platforms (Gonidis et al., 2013).

Standardisation Approach This approach intends to standardise interfaces in each service level of cloud computing, so that cloud applications and resources are provisioned, deployed and managed independently of specific platform environments. It is an efficient approach to accomplish cloud integration (cloud interoperability and portability), but it is very complicated for different cloud platforms to agree on a common standards.

Intermediation Approach This approach intends to provide an intermediate layer (middleware service or platform) that hides the proprietary APIs of cloud providers. It achieves that by dealing with several vendor-specific connection protocols and APIs, and vendor-specific provisioning interfaces as well as all stages of the software development lifecycle. As the integration challenge is raised, some recent efforts such as SimpleCloud and mOSAIC (Di Martino et al., 2015) have attempted to mask the API heterogeneity between different cloud platforms by providing uniform, multi-provider compatible APIs. Libraries such as jClouds enable access, provisioning and management of resources from different clouds, and mOSAIC offers the developers an abstraction from native APIs by implementing several API layers (Gonidis et al., 2013). Therefore, this approach provides a quick and easy way to have access to different clouds supported by the selected toolkit.

2.5.2 Cross-Cloud Workflow Migration

The migration process here aims to migrate the workflow completely or part of it (in terms sub-workflows or analytic activities) from one cloud system to another, targeting several optimisation requirements such as improving performance, reducing execution cost and time, and achieving specific QoS features. Figure 2.3 shows the classification of cross-cloud migration approaches for workflow. As seen from this figure, the three approaches for migrating workloads (i.e. workflow and its analytic activities) between different execution environments are workflow abstraction-based, cloud broker-based and container-based approaches.

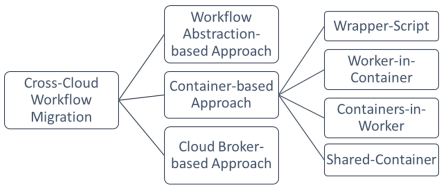


Figure 2.3: Classification of Cross-Cloud Migration Approaches for Workflow

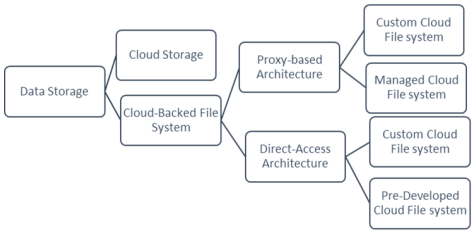


Figure 2.4: Classification of Data Storage Approaches for Workflow

Workflow Abstraction-based Approach This approach aims to describe abstract data-intensive workflows, enabling the portability of these workflows across diverse execution platforms. The abstract model is used to define data-intensive workflow and removing the details of target execution platforms and the steps of data handling (Filgueira et al., 2016) (Filguiera et al., 2017). Makeflow (Albrecht et al., 2012), Asterism DIaaS (Filgueira et al., 2016) and dispel4py (Filgueira et al., 2015) (Filguiera et al., 2017) are examples of workflow abstraction models that are not mainly designed to support the heterogeneity and dynamism of big data workflows, however, they can be elaborated in abstraction of those workflows.

Cloud Broker-based Approach This approach provides the ability to run workflow applications in intercloud environments. It acts as mediator among users of workflow and providers of cloud systems, helping in the selection of target cloud(s), accessing this/those cloud(s) and achieving user-defined SLA and QoS requirements (Jrad et al., 2012) (Jrad et al., 2013).

Container-based Approach This approach exploits containerisation technology (e.g. Docker, udocker (Gomes et al., 2018), Singularity (Kurtzer et al., 2017)) to provide the ability to quickly and efficiently build and deploy workflows (sub-workflows or workflow activities) across cloud computing systems by encapsulating compute resources and delivering a user-defined execution environment (Gerlach et al., 2014) (Qasha et al., 2016) (Zheng and Thain, 2015). A container packs only the libraries and packages (i.e. full software stack) needed by sub-workflow or workflow activity (Qasha et al., 2016). By doing that, the workflow portability and migration are improved, allowing seamless and agentless migration of workflows across diverse cloud infrastructures.

2.5.3 Resource Provisioning

Resource provisioning aims to select and provision the cloud resources that will be used to execute the tasks (i.e. big data workflow fragments or tasks). There are two following approaches to resource provisioning.

Static Resource Provisioning Approach This approach takes the decision of provisioning virtual resources that are required to run workflow fragments or tasks before the

execution of workflow. It is not able to dynamically scale resource in or out (Rodriguez and Buyya, 2017). The provisioned resources are fixed, and they are the only resources available during the whole period of workflow execution. Thus, such an approach is suitable to be used in cases where the demand of the workflow is predicted and fixed in term of resources.

Dynamic Resource Provisioning Approach In contrast, this approach takes the decision of provisioning resources during the execution of workflow or at runtime. It decides which resource types and configurations are most suitable, and when to add or remove resources according to the demands of the workflow. In other words, this approach is taking all decisions or refining initial ones at runtime and determining which virtual resources need to keep running and active, which resources should be provisioned and which resources from the provisioned resources should be deprovisioned as the workflow execution progresses. This approach aims to avoid under-provisioning because of its implication on performance (lowers performance) and over-provisioning because of its implication on cost and system utilisation (increase the cost and lowers system utilisation).

2.5.4 Resource Volatility

In any environment, there is a possibility of losing these resources or the state of analytical processes executed by big data programming framework at any time due to different failures. Mitigating such failures need to be carried out at different levels (VM level, big data programming framework level and workflow task level). For each level, a corresponding approach is needed to mitigate those failures that occur at this level, achieving that without ignoring the consideration of resource consumption and performance efficiency. Therefore, there are three level-based approaches: at VM-level, at data processing level and at workflow-level.

- VM-level Mitigation Approach: This approach aims to mitigate the failure and the loss of the state of virtual machine in terms of data stored in RAM and/or non-persistent storage. Examples of techniques under this approach are replication approaches based on active/active or active/passive mode. Checkpointing is a common technique that can be used to save or replicate the state of VM (periodically or on-demand) and then mitigate failures by recovering from stored or replicated state (Souza et al., 2018) (Dong et al., 2013). VM workload consolidation-based fault-tolerance technique is another technique used to improve the VM's reliability (Li et al., 2017).

- Big Data Processing Framework-level Mitigation Approach: This approach aims to mitigate the failure and the loss of the state of computational units/processes within big data processing system (such as with Apache Hadoop is the sense of losing the state of mapper and reducer processes which we cannot capture at VM level). Examples of techniques used to recover from the failures of data processing tasks are byzantine fault tolerance (in MapReduce (Costa et al., 2011)), replication-based fault tolerance

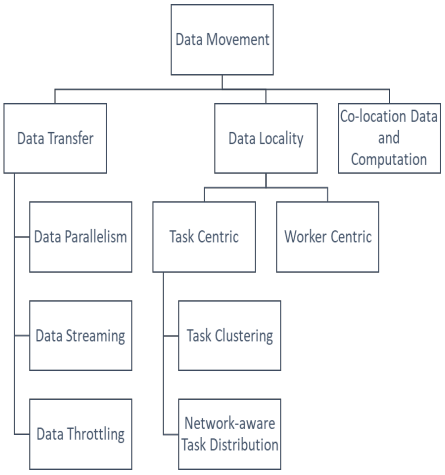


Figure 2.5: Classification of Data Movement Approaches for Workflow

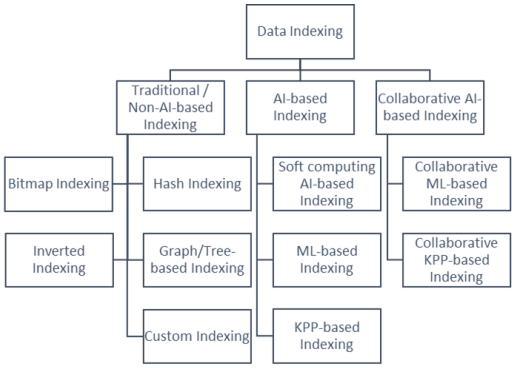


Figure 2.6: Classification of Data Indexing Approaches for Workflow

(in MapReduce (Liu and Wei, 2015)) and rollback recovery (in dataflow systems (Isard and Abadi, 2015)).

- Workflow task-level Mitigation Approach: This approach aims to mitigate the failure and the loss of workflow tasks including the loss of analytic computation completed so for, which may incur additional cost or delay in execution with the best case. Workflow task-level techniques (either reactive or proactive) can be to handle task failures occurring during the execution of workflow.

2.5.5 Data Storage

Big data workflow comprises of a set of data-intensive tasks, which communicate using large files (large datasets). These large files should be stored in the cloud since the execution of big data workflows will be carried out in the cloud, and be passed among workflow tasks using data movement techniques or shared storage systems (Cafaro and Aloisio, 2011). Figure 2.4 shows the classification of data storage approaches for workflow.

Cloud Storage This is a storage service offered by cloud providers. This approach requires the workflow management system to manage data on the cloud storage service (Cafaro and Aloisio, 2011). However, the reliability of data stored in cloud storage system could be an issue with this approach (Nachiappan et al., 2017).

Shared Cloud-backed File System It intends to deploy shared file systems in the cloud (Cafaro and Aloisio, 2011), where the backend can be single cloud (by utilising single cloud storage service) or cloud-of-clouds (by utilising multiple cloud storage services) . It resolves the storing data problem in a generic way and follows either a proxy-based architectural model or direct-access architectural model (Bessani et al., 2014). The descriptions of these models are as follows:

- Proxy-based Model − In this model, the proxy is implementing the core functionality of the file system and interacting with cloud storage in order to store and retrieve files (Bessani et al., 2014). With the single file system, the single point of failure and performance bottleneck are issues with this model (Bessani et al., 2014), while the parallel file system addresses those issues. The file system following this model can be:
  - Custom Cloud File System − The aim here is to build a custom shared file system for workflow tasks. For example, a file system can be hosted in extra VM provisioned from a particular cloud platform/infrastructure, and the other provisioned VMs, i.e. worker nodes can mount such a file system as a local drive/volume (Cafaro and Aloisio, 2011). In addition, parallel file systems can be hosted by several VMs in case better performance is required (Cafaro and Aloisio, 2011).
  - Managed Cloud File System − The aim here is to select and use one of the shared file system options offered by cloud providers.
- Direct-access Model − In this model, there is no proxy and the access to the cloud storage is direct. Also, with this model, the single point of failure is no longer an issue, but it becomes hard to offer file sharing in a controlled manner since the convenient rendezvous point for synchronisation is missed (Bessani et al., 2014). The file system following this model can be:
  - Custom Cloud File System − The aim here is to build a custom shared file system for workflow tasks without the interposition of a proxy.
  - Pre-developed Cloud File System − The aim here is to select and use an existing shared file system.

2.5.6 Data Movement

By moving the execution of big data workflow to the cloud, the working datasets should be moved to the cloud as well. These datasets are large datasets and moving or transferring them is an expensive task. In the literature, several research works have been proposed various approaches to tackle the problem of data movement for data-intensive workflows. The classification of data movement for workflow is depicted in Figure 2.5.

Data Transfer This approach intends to transfer data with minimal data transfer time. The heterogeneity and instability of the cloud network affect this transfer (Tudoran et al., 2016). The following are three different techniques to achieve lowest data transfer time (Pandey and Buyya, 2012):

- Data Parallelism − The ability of a service to process data chunks in parallel with minimal performance loss. Such ability includes the processing of independent data on various compute resources.

- Data Streaming – This technique intends to enable data transport among workflow fragment/tasks through the use of data streams. That allows support for high-throughput and low latency.

- Data Throttling – This technique intends to determine and control the arrival time and the rate of data transfer as opposed to the movement of data from one place to another as quickly as possible. As an alternative of transferring data to a task, this technique can be used to delay data transfer or transfer data using lower capacity links in order to allocate resources to serve other crucial tasks.

Data Locality Since the working datasets for big data workflow are huge, moving those datasets among compute resources provisioned from multiple clouds is costly and time-consuming. This approach aimed at minimising data movement by means of moving the computation in proximity of data. The different techniques to exploit such approach are as follows:

- Task Centric – This technique aims to move workflow tasks towards data without considering the interest of workers. The locality of data is exploited by schedulers to map tasks to compute resources in which tasks are being executed on compute resource that is in or close to the location of data. Task clustering is a method that aims to group small workflow tasks together as one executable unit for eliminating data movement overhead (and by the way removing the overhead of executing those small tasks). By doing grouping of tasks, the intermediate results generated by each grouped task remains in the same virtual resource (i.e. VM), which allows other grouped tasks to locally access the result. A special case of task clustering is spatial clustering. With this method, a workflow task is created by relying on the spatial relationship of files in datasets (Pandey and Buyya, 2012). It groups workflow tasks into clusters based on spatial proximity, where each cluster contains a subset of tasks and is assigned to one execution site. Network-aware task distribution is a method exploited by a scheduler to mix data localisation and geo-distributed datacenter data transfer (network bandwidth) requirements to tackle the data movement problem for a large-scale application whose data has been collected and stored across geo-distributed datacenters and is subject to certain constraints (e.g. GDPR) (Hung et al., 2015) (Hu et al., 2016) (Jin et al., 2016) (Convolbo et al., 2018) (Chen et al., 2018b).

- Worker Centric – This technique aims not only to exploit the locality of data, but also to take into consideration the interests of workers on executing computation. The idle worker takes the intuitive and expresses its interest to execute a workflow task, in that case, the scheduler chooses the best task for this worker by exploiting locality in data accesses.

Co-location of Data and Computation Instead of moving data to compute nodes or bringing computation to the data, co-locating data and computation is a viable solution; it addresses the data problem as part of the resource utilisation strategy. Thus, this approach aims to combine compute and data management for tackling the problem of data movement to minimise the overhead and scalability for forthcoming exascale environments for to achieve better resources utilisation.

2.5.7 Data Provenance

As mentioned earlier,the provenance data for big data workflow represents the execution behavior of such workflows, which allows tracing the data-flow generation (Costa et al., 2014). To provenance data, there are two following approaches based on granularity level (Glavic, 2014).

Coarse-grained Provenance It is control-flow-based approach that does not peer into data-flow inside transformations and handles them as black boxes, so that for a given transformation, it records the elements of data that are inputs and outputs of such a transformation. For instance, with word count transformation and by considering documents as single data units, this approach deliberates all documents as a pair (w, c) provenance. The graph structure is usually used to represent such information in which data elements are linked to provenance transformations that generated or consumed those elements.

Fine-grained provenance It is data-flow-based approach that peers into data-flow inside transformations to provide insight information. In other words, this approach exposes the transformation processing logic as a result of modelling the significant parts of inputs in the derivation of a specific output data element. For instance, with word count transformation and by considering documents as single data units, this approach deliberates input documents that contain the word w as provenance of a pair (w, c).

2.5.8 Data Indexing

The aim of data indexing is to accelerate data access as well as data query operations but it comes at an extra cost for both data index creation operations and data writing operations, and additional storage space required for storing these indexes. Various indexing approaches have been reviewed and investigated for big data requirements in the literature (Gani et al., 2016) (Adamu et al., 2016) (Cai et al., 2017). The classification of data indexing for big data in big data workflow is depicted in Figure 2.6.

Traditional / Non-AI-based Indexing With this approach, neither the meaning of the data element nor the relationship among words is included in the index formation. That means the formation of indexes is dependant on the cover-known patterns (i.e. most searched

and retrieved data elements) in a given dataset. Hash indexing is an efficient strategy for data access and retrieval in a high-dimensional data context as it is able to detect duplication of data in a big dataset. Bitmap Indexing is a strategy that uses bitmap data structure for storing data and then retrieving it quickly. It works nicely with low-cardinality columns as well as it is considerably being appropriate for big data analysis along with low data storage space. Graph/Tree-based indexing strategy is a strategy that uses the index of more complex data structures to make data indexes to enhance the performance since the bitmap data structure is feeble in transaction processing. Examples of such data structures used for indexing data are B-Tree, B+-Tree, R-Tree, Log-Structured Merge (LSM)-Tree, bLSM (LSM with B-Trees and log structured methods). In case of storing the big data of workflow application by using SQL model with many relational database systems and/or NoSQL model with Cassandra, BigTable or/and HBase, this approach is followed but with different data strictures. Indeed, many relational database systems used B-Tree data structure while aforementioned NoSQL database management systems are used LSM-Tree data structure (Tan et al., 2014) to support data indexing. Inverted Indexing strategy intends to enhance the full-text searching capability by the use of an inverted index data structure to store the mapping of content (e.g. numbers, word sequences) to its location in document database. Custom indexing strategy intends to create multiple field indexing by replying on either random or user-defined indexes. Generalized Inverted Index (GIN) and Generalized Search Tree (GiST) are two types of custom indexing (Adamu et al., 2016).

AI-based Indexing This approach is able to discover unknown big data behaviour by utilising a knowledge base, providing efficient data indexing, and thus effective data search and retrieval. However, it needs more time compared with a non-AI indexing approach to answer the search query in general. Soft computing AI-based indexing techniques blend fuzzy set and neural computing methods for indexing data, while Machine Learning (ML)-based indexing techniques improve data indexing by utilising machine learning methods such as manifold learning. Knowledge Representation and Reasoning (KRR)-based indexing achieves that using semantic ontology.

Collaborative AI-based Indexing This approach enhances the accuracy of data indexing and the efficiency of search by relying on collaborative artificial intelligence, aimed at providing greater cooperative data indexing solutions. With this approach, collaborative ML-based indexing and collaborative KRR-based indexing methods are provided that relate individual and cooperative decision-making to index big data.

2.5.9 Workflow Specification Language

Workflow specification language is used to describe the structure of workflow and its tasks to allow interpreting and executing the specification. There are two approaches that we can consider here for specification language: generic and custom. In generic approach, the generic

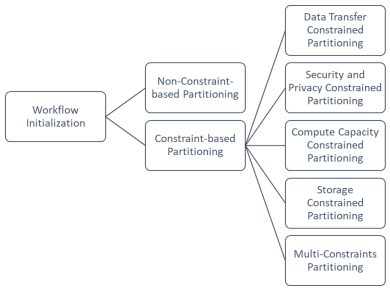


Figure 2.7: Classification of Workflow Initialisation Approaches

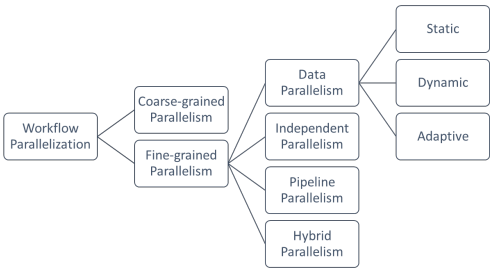


Figure 2.8: Classification of Workflow Parallelisation Techniques

(general-purpose) workflow specification language is properly selected and used to describe the big data workflow. In custom approach, the new workflow specification language is designed to describe big data workflows. Such a specification language limits the portability and scalability capabilities for workflow across a variety of execution environments.

2.5.10 Workflow Initialisation

Workflow initialisation aims to partition (with or without constraint) a workflow into fragments to parallelise the execution of those fragments over provisioned compute resources. Since big data workflow is composed of data-intensive tasks, parallelising the execution of these tasks needs partitioning of such workflow into fragments. The approaches of workflow initialisation can be classified as either non-constraint-based partitioning or constraint-based partitioning. This classification is depicted in Figure 2.7.

Non-Constraint-based Partitioning This approach decomposes a workflow into smaller fragments to allow distribution of those fragments among compute resources for parallel execution. It considers the task and data dependencies within the workflow, and avoids cross dependency, no other constraints are taken into account. Thus, the decision of partition is made based on task and data dependencies and not based on the capacity of compute resources or the cost of data movement.

Constraint-based Partitioning This approach partitions a workflow into smaller fragments, taking into consideration the defined constraint, to allow distribution of those fragments among compute resources for parallel execution. It not only considers the task and data dependencies within the workflow, and avoid cross dependency, but also any other constraint that is defined. There are five following techniques to support constraint-based partitioning.

1. Data Transfer Constrained Partitioning − This technique aims to minimise the amount of data to be moved among fragments of a workflow (Liu et al., 2014). By considering the cost of transferring data between fragments that will be executed in one site or

multisite as a partitioning constraint, the workflow will be decomposed in such a way that minimises data transfer so as to reduce the total execution time.

2. Security and Privacy Constrained Partitioning − This technique aims to partition a workflow into fragments under security and privacy restrictions. For instance, a workflow may contain a critical activity that requires execution to be done at a trusted cloud site, so this workflow will be partitioned in such a way that this activity and its following activities for processing output data must be designated to the same fragment, and the others will be designated to another fragment.

3. Compute Capacity Constrained Partitioning − This technique partitions a workflow into fragments according to compute resource configurations. The different configurations of compute resource in one cloud or heterogeneous multisite cloud configurations can be used to adapt workflow partitioning (Liu et al., 2014). For example, some tasks of a workflow may need more computing capacity than other tasks, so that those tasks will be assigned to available compute-intensive resources or to the cloud site that has more compute capacity.

4. Storage Constrained Partitioning − This technique aims to respect storage constraints during partition of a workflow into fragments (Chen and Deelman, 2011).

5. Multi-Constraints Partitioning − This technique aims to respect multiple factors or constraints in the process of partitioning a workflow.

2.5.11 Workflow Parallelisation and Scheduling

Following the classification of workflow parallelisation techniques presented by (Liu et al., 2015b), the two parallelisation techniques based on the level of parallelism are coarse-grained parallelism and fine-grained parallelism (Liu et al., 2015b). This classification is depicted in Figure 2.8.

Coarse-grained Parallelism This approach achieves parallelisation at the level of workflow. It is crucial to meta-workflow execution or parameter sweep workflow execution. For meta-workflow, this technique parallelises the execution of independent sub-workflows composed of such workflow by submitting them to corresponding workflow engines. In a parameter sweep workflow execution, each set of input parameter values results in an independent sub-workflow.

Fine-grained Parallelism This approach achieves parallelisation at activity level within a workflow or a sub-workflow, where different activities will be executed in parallel. At this level, there are different types of parallelism to handle within an activity and between activities. For parallelism within an activity, data parallelism is used; for parallelism between

activates, independent parallelism and pipeline parallelism are used; and for higher degrees of parallelism, hybrid parallelism is used. Following are their description (Liu et al., 2015b):

- Data Parallelisation − This type handles parallelism within an activity. To achieve such parallelism, it needs to have various tasks perform the same activity and each one of them processes different chunks of input data in a various compute node. Thus, the resultant data is partitioned since the input data is already partitioned. This partitioned result (output data) could be input data for data parallelism for the next activities or be combined in order to produce single result. This type of parallelism can be static, where the number of data portions is fixed and specified prior the execution, dynamic, where the number of data portions is identified at runtime, and adaptive, where the number of data portions is automatically modified to the execution environment.
- Independent Parallelism − This type handles parallelism between independent activities of a workflow. To achieve such parallelism, workflow should have at least two or more independent fragments of activities and the activities of each fragment have no data dependencies with activities of other fragments, as well as those independent activities need to be identified in order to be executed in parallel.
- Pipeline Parallelisation − This type handles parallelism between dependent activities. These activities with a popular type of relationship among activities (i.e. producer-consumer relationship) can be parallel executed in pipeline fashion, where the output of one data portion of one activity is the input of the following dependent activities. By exploiting this type of parallelism, the consumption of data portions is performed as soon as those portions are ready.
- Hybrid Parallelism − This type combines three types of parallelism in order to achieve higher degrees of parallelism. It applies data parallelism within each activity, then independent parallelism between independent activities and lastly pipeline parallelism between dependent activities.

After parallelising the activities of big data workflow, these activities should be scheduled on cloud resources for execution. Figure 2.9 shows the classification of workflow scheduling techniques. The techniques of workflow scheduling can be categorised into push-based and pull-based scheduling.

Push-based Scheduling This technique allows scheduling tasks of workflow among compute resources by pushing them to available resources. The scheduler maps workflow tasks to resources according to the generated scheduling plan. By following task-centric, the scheduling techniques are as follows:

- Static scheduling − This technique generates and assembles schedules that allocate all tasks of workflow to compute nodes prior to the execution of workflow and these schedules (i.e. scheduling plan) are strictly observed during the whole execution (Bux and Leser, 2013) (Liu et al., 2015b). Since the scheduling decision is made before

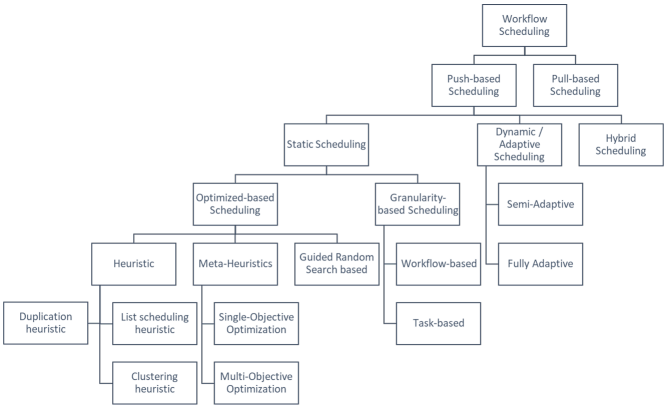


Figure 2.9: Classification of Workflow Scheduling Techniques

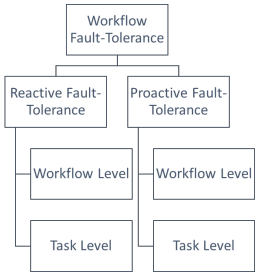


Figure 2.10: Classification of Workflow Fault-Tolerance Technique

execution of workflow, this technique produces little overhead at runtime. It is efficient and achieves good results when execution environment experiences little change, i.e. controllable or homogeneous compute environments. With execution environments that vary greatly, it is very hard to achieve load-balance, and with variations in resource performance, the overall execution time will be strongly impaired (Bux and Leser, 2013) (Liu et al., 2015b). There are various scheduling techniques which can be classified into the following categories:

- Granularity-based scheduling − In this category, the process of scheduling workflow is based on granularity level. The techniques/methods are (Liu et al., 2015b):
  - Workflow-based − This technique maps the partitioned fragments of workflow to compute resources. It is a preferable technique used for data-intensive applications because the overhead of transferring data between fragments is less than transferring data between tasks.
  - Task-based − This technique maps the tasks of workflow directly to compute resources.
- Optimised-based scheduling − In this category, the scheduling problem is considered as an optimisation problem. The techniques/methods are (Wu et al., 2015) (Liu et al., 2015b):
  - Heuristic: There are three types of heuristics that have been proposed. List scheduling heuristic constructs a scheduling list for tasks that will be scheduled by appointing those tasks some priorities, and then sorting them in accordance with the assigned priorities, then performing "task selection" and "resource selection" steps recurrently until all tasks in the directed acyclic graph are scheduled, where in the "task selection" step, the head of the scheduling list is selected (i.e. first task) and in the "resource selection" step, the task is allocated to the selected resource. Clustering heuristic focusses on optimising the time of transmission among data dependent tasks. A gen-

eral clustering heuristic comprises of two phases, which are clustering (to map tasks to clusters) and ordering (to order tasks that belong to the same cluster). Similarly, duplication heuristics focus on optimising the transmission time using duplication of tasks.

  - Meta-Heuristics − The scheduling algorithm uses a global search oriented meta-heuristic to find a very good solution quickly and efficiently. The most used meta-heuristic for task scheduling problems is the genetic algorithm. Other meta-heuristics are investigated and used such as ant colony optimisation and particle swarm optimisation. There are two types of meta-heuristics algorithms: (1) single-objective optimisation for optimising the value of single objective function either by minimising or maximising it, and (2) multi-objective optimisation for more complex problems with more than one conflicting objective that it is obliged to optimise simultaneously, where it provides alternative optimal solutions by trading-off between those objectives (Talbi et al., 2012).
  - Guided random search based − This technique schedules tasks randomly.

- Adaptive/Dynamic scheduling − This technique generates a scheduling plan that maps workflow tasks to compute resources at runtime by monitoring execution infrastructure (Liu et al., 2015b) (Bux and Leser, 2013). Such a plan is adjusted continually during the execution of workflow according to the perceived changes. It is a suitable technique for use when workflow tasks exist in a highly dynamic environment or when the amount of work for those tasks is hard to estimate. This technique can be either:

  – Semi-Adaptive − In this type, the scheduler maps tasks to compute resources during workflow execution in accordance to the on-line performance statistics of the resource.

  – Full-Adaptive − In this type, the scheduler schedules tasks onto compute resources during workflow execution in accordance to the on-line performance statistics of the resource, plus specific task requirements and characteristics. For example, a full-adaptive scheduler may map a workflow task with a high degree of parallelism on a compute resource that has multiple threads.

- Hybrid scheduling − This approach combines static and dynamic scheduling to gain the advantages of both in order to provide better performance than just using one or the other (Liu et al., 2015b). For instance, the static scheduling can be used to schedule part of workflow tasks (e.g. there is enough information for them) and the remaining tasks can be scheduled at runtime using dynamic scheduling (Liu et al., 2015b).

Pull-based Scheduling This technique intends to exploit the interest of worker/node in scheduling a task when it is idle by allowing the worker to request from a scheduler to

schedule a task and the scheduler pulls the best task for this worker from among available tasks and maps such tasks to it.

2.5.12 Workflow Fault-tolerance

Big data workflow composed multiple data-intensive tasks and the execution of those tasks is usually a lengthy process, so that failures or errors could happen at any time during the execution period. Workflow fault-tolerance intends to handle failures occurring during the execution of workflow and assure its availability and reliability (Liu et al., 2015b). Figure 2.10 shows the classification of workflow fault-tolerance techniques.

Reactive Fault-Tolerance This technique aimed at minimising the impact of failures after their occurrence. To achieve that, there are numerous fault-tolerance techniques such as checkpoint/restart, replay (replication) and retry (task resubmission). This technique can resolve the faults at either:

- Workflow level − As th name suggests, detecting failures is carried out at workflow level, similar to application level. Thus, it deals with the failures of the execution of sub-workflows or workflow fragments by resubmitting the affected fragment.
- Task level − At this level, task failures are detected after perceiving failures and then being resolved.

Proactive Fault-Tolerance This technique avoids waiting until the failures or errors occur and then recovering from them by foreseeing the failure and proactively substituting those components that have been suspected from those other components that are working properly (Liu et al., 2015b). It can resolve the faults at either:

- Workflow level − The aim here is to predict the failures at workflow level, in other words, it predicts the failures of sub-workflows or workflow fragments. For example, if an error is predicted for a given workflow fragment, it is replaced proactively from other working fragments.
- Task level − The focus here is on the tasks of workflow, where it predicts task failures and replaces them proactively from other working tasks.

2.5.13 Workflow Security

In the context of big data workflow, securing big data is not the whole story, it is considered as a part in preserving workflow security. The other part is guaranteeing the security of workflow logic and computation. Workflow security aims to secure the data-intensive tasks, which process and generate vast amounts of data. It is intended to ensure the security and integrity of the logic of operations. As big data workflow is an emerging research topic, there

is a very limited research on this problem, and therefore preserving the security of this type of workflow is still an open issue (see Section 2.7). However, some existing techniques that can be utilised are Multicloud architectures, replication-based techniques (with or without trust component called verifier) and task selection techniques (Bohli et al., 2013) (Shishido et al., 2018). In addition, there are other research works have proposed Multicloud security framework for data intensive workflows such as (Demchenko et al., 2017) and secure workflow deployment technique on federated clouds such as (Wen et al., 2017).

Table 2.2 shows the example work(s) for each classification level (approach) in the presented research taxonomy.

Table 2.2: Exemplar works under each classification level in the presented research taxonomy

Taxonomy	**Approaches**	**Exemplar Work**
Cloud Platform Integration	Standardisation Approach	Standardisation bodies: NIST, DMTF, SNIA and ITU-T (Peoples et al., 2013) Major open standards: OVF (by DMTF), CDMI (by SNIA) and OCCI (through OGF) (Zhang et al., 2013)
	Intermediation Approach	SimpleCloud, DeltaCloud, Libcloud, jClouds and mOSAIC
Cross-Cloud Workflow Migration	Workflow Abstraction-based Approach	Makeflow (Albrecht et al., 2012), Asterism DIaaS (Filgueira et al., 2016) and dispel4py (Filgueira et al., 2015) (Filguiera et al., 2017)
	Cloud Broker-based Approach	Cloud service broker (Jrad et al., 2012), STRATOS (Pawluk et al., 2012) and Broker-based framework for workflow applications (Jrad et al., 2013)
	Container-based Approach	Skyport (Gerlach et al., 2014), Containerisation strategies within workflow system (Zheng and Thain, 2015), TOSCA-based platform (Qasha et al., 2016), Asterism (Filgueira et al., 2016) and CoESMS (Kaur et al., 2017)
Resource Provisioning	Static Resource Provisioning	AROMA (Lama and Zhou, 2012)
	Dynamic Resource Provisioning	Cost-aware and SLA-based algorithms (Alrokayan et al., 2014), RPS (Cao et al., 2016), Resource provisioning method (Todd Jr et al., 2017) and Data-aware provisioning algorithm (Toosi et al., 2018)
Resource Volatility	VM-level Mitigation Approach	COLO (Dong et al., 2013), VM workload consolidation-based fault-tolerance technique (Li et al., 2017) and Hybrid adaptive checkpointing technique (Souza et al., 2018)

	Big Data Processing Framework-level Mitigation Approach	MapReduce Online (Condie et al., 2010), BFT MapReduce technique and prototype (Costa et al., 2011), Falkirk wheel (Isard and Abadi, 2015), Mapreduce replication-based fault-tolerance technique (Liu and Wei, 2015) and Checkpointing & confined and replica recovery techniques for dataflow systems (Xu et al., 2017)
	Workflow Task-level Mitigation Approach	Workflow task-level techniques in workflow fault-tolerance classification can be used here
Data Storage	Cloud Storage	Amazon Cloud Storage (S3) , Microsoft Azure Data Storage , Google Cloud Storage , Rackspace Database Service and Rackspace Cloud Block Storage
	Shared Cloud-backed file system	Gfarm (Mikami et al., 2011), BlueSky (Vrable et al., 2012), DepSky (Bessani et al., 2013), WaFS (Wang et al., 2014c), Týr (Matri et al., 2016) and Faodel (Ulmer et al., 2018)
Data Movement	Data Transfer	Online parallel compression framework (Bicer et al., 2013) and Data throttling technique in the proposed system (Mon et al., 2016)
	Data Locality	Two-stage data placement method (Zhao et al., 2015a), Task placement method (Ebrahimi et al., 2015), Heuristic data placement method (Zhao et al., 2016a), Clustering method based task dependency (Mon et al., 2016), GEODIS (Convolbo et al., 2018) and Fair job scheduler (Chen et al., 2018b)
	Co-location Data and Computation	DPPACS (Reddy and Roy, 2015), Task assignment method (Zhao et al., 2016b) and DACS (Hassan et al., 2017a, Chapter 18)
Data Provenance	Coarse-grained Provenance	Stream provenance method (Vijayakumar and Plale, 2007) and Workflow provenance management in WorkflowDSL (Fernando et al., 2018)
	Fine-grained Provenance	RAMP (Park et al., 2011), On-the-fly provenance tracking technique (Sansrimahachai et al., 2013), Ariadne (Glavic et al., 2014), Big data provenance techniques (Chen, 2016), Titian (Interlandi et al., 2018) (Interlandi and Condie, 2018) and DfAnalyzer (Silva et al., 2018)
Data Indexing	Traditional / Non-AI-based Indexing	Bitmap (Wu et al., 2010), Diff-Index (Tan et al., 2014), Inverted index pruning approach (Vishwakarma et al., 2014), UQE-Index (Ma et al., 2012), GIN , Metadata index and search system (Yu et al., 2014), SpatialHadoop (Eldawy and Mokbel, 2015)
	AI-based Indexing	GRAIL (Yıldırım et al., 2012) and Semantic indexing technique (Rodríguez-García et al., 2014)

	Collaborative AI-based Indexing	Collaborative semantic technique (Gacto et al., 2010), Collaborative learning (Fu and Dong, 2012) and Collaborative filtering technique (Komkhao et al., 2013)
Workflow Specification Language	Generic Approach	YAWL (Van Der Aalst and Ter Hofstede, 2005) and CWL (Amstutz et al., 2016)
	Custom Approach	WorkflowDSL language (Fernando et al., 2018)
Workflow Initialisation	Non-Constraint-based Partitioning	Workflow partitioning method (Chen and Deelman, 2012a)
	Constraint-based Partitioning	Workflow partitioning based on storage constraints (Chen and Deelman, 2011), PDWA (Ahmad et al., 2014) and I-PDWA (Ahmad et al., 2017)
Workflow Parallelisation	Coarse-grained Parallelisation	Workflow-level parallelism in Globus Genomics system (Bhuvaneshwar et al., 2015) and Type-A workflow execution algorithm (Mohan et al., 2016)
	Fine-grained Parallelisation	Online parallel compression framework (Bicer et al., 2013), Type-B workflow execution algorithm (Mohan et al., 2016)
Workflow Scheduling	Push-based Scheduling	SLA-Based resource scheduling (Zhao et al., 2015b), Dynamic fault-tolerant scheduling method (Zhu et al., 2016), Tree-to-tree task scheduling technique (Zhao et al., 2016b), Stable online scheduling strategy (Sun and Huang, 2016), T-Cluster algorithm (Mohan et al., 2016), Elastic online scheduling (Sun et al., 2018), GEODIS (Convolbo et al., 2018) and Fair job scheduler (Chen et al., 2018b)
	Pull-based Scheduling	Data-aware work stealing technique (Wang et al., 2014b) (Wang et al., 2016)
Workflow Fault-Tolerance	Reactive Fault-Tolerance	Fault-tolerance scheduling algorithm (Poola et al., 2014), Fault-tolerant scheduling technique (FASTER) (Zhu et al., 2016) and Fault-tolerance scheduling heuristics (Poola et al., 2016)
	Proactive Fault-Tolerance	FTDG (Sun et al., 2017)

2.6 Systems With Big Data Workflow Support

There are several platforms that can be extended or have capability to support big data workflows. As discussed before, scientific workflow management systems also require coping with large volumes of data, hence in recent years many of them have been extended to support big data applications. Similarly, there are platforms/systems that have been designed specifically for orchestrating the execution of big data applications such as YARN. In this section, we survey these systems respectively.

2.6.1 Scientific Workflow Systems with Big Data Extensions

Since the demand for data-intensive scientific workflow has increased and with the emerging of big data technology, several research works have extended the functionalities of existing SWMSs with data-intensive capabilities in order to enable big data applications in SWMS. We summarise and compare those research works (aka data-intensive scientific workflow management systems) in Table A.4 of Appendix A.2.

2.6.2 Big Data Application Orchestrator

There are existing workflow tools that can be integrated with Hadoop to support MapReduce workflows, which are Luigi, Linkedin Azkaban, Apache Oozie and Airflow. These tools are specific-purpose workflow managers that do not need to support the dynamism and heterogeneity of big data workflows. Rachel Kempf (lKempf, 2017) compared these tools and highlighted their features. In the same context, Garg et al. (Garg et al., 2018) reviewed and compared the current orchestration tools for big data. Some of tools reviewed in this research book chapter are also specific-purpose workflow managers. Accordingly, there are mainly three big data orchestrating systems that can be extended for big data workflow management. They are Apache YARN, Apache Mesos and Amazon Lambda. The details of each of these platforms can be studied from Appendix A.3. As these platforms can be extended for big data workflow management, we discuss their capabilities against the challenges taxonomy that shown in Figure 2.2.

Cloud Resource Management Challenge For Apache YARN and Apache Mesos, the number and types of compute resources that will be allocated to workflow tasks need to be pre-selected as well as the configuration of these resources being determined. Therefore, these systems use limited resource provisioning since the provisioned resources are pre-determined and limited during the workflow execution (the only available resources for this workflow). Of course, both of them take into consideration managing all the available compute resources in all machines in the managed cluster. For AWS Lambda, the compute resource is determined based on the amount of memory , so that the amount of memory allocated to the Lambda function needs to be pre-determined, and AWS Lambda allocates the power of CPU proportional to that amount by using the same ratio as a general purpose Amazon EC2 instance type (Amazon, 2017b). The scaling of compute capacity is done dynamically by AWS Lambda in accordance to traffic load(Amazon, 2017b).

Data Management and Storage Challenge In Apache YARN, the use of HDFS is to store large amounts of data on cheap clusters and to provide high-performance access to that data across the cluster. Thus, Apache YARN utilises a cloud-backed file system approach that allows it to deploy a shared file system in the cloud for a workflow. Moreover, for data movement, Apache YARN exploits data locality, and since the RM is a central authority

and has a global view of cluster resources, it can enforce locality across tenants (Vavilapalli et al., 2013). On the other hand, Apache Mesos offers persistent volumes to store data. The persistent volume can be created once a new task is launched, exists outside the sandbox of a task, provides exclusive access to a task by default and will persist on the slave node even after the task finishes or dies. Shared persistent volumes is also supported by Apache Mesos in order to allow sharing of a volume between multiple tasks operating on the same node.

Data Security and Privacy Challenge Authentication is supported by all systems, where Apache YARN uses Kerberos authentication, Apache Mesos uses a factor authentication approach, i.e. a challenge-response protocol (CRAM-MD5), which is essentially single-factor authentication. AWS Lambda also supports a factor authentication approach, i.e. multi-factor authentication. In addition, controlling the access to resources and services is provided by the reviewed systems. Apache YARN supports coarse-grained access control, while Apache Mesos supports some extent fine-grained access control, and AWS Lambda provides fine-grained access control via AWS IAM. Moreover, to encrypt data and for communication data to remain private and integral, Apache YARN and Apache Mesos use Secure Sockets Layer (SSL) and Hypertext Transfer Protocol Secure (HTTPS), where SSL uses public-key cryptography at first and then symmetric cryptography for the rest of computation to encrypt the transmitted data.

Workflow Scheduling Challenge Apache YARN offers a single centralised scheduler to schedule competing workflow tasks among compute resources in the cluster. Therefore, Apache YARN uses a push-based approach with static scheduling. On the other hand, in Apache Mesos, the distributed two-level scheduling mechanism that lets the framework either accept the offer, or reject it. After the offered resource(s) is accepted by the framework, this framework passes the task description to Apache Mesos, which launches the tasks on the corresponding agents [1] using push-based approach. This scheduling mechanism is called "resource offers".

Workflow Fault-Tolerance Challenge In Apache YARN, the RM detects and recovers from its own failures, where with work-preserving RM restart, the running applications will not lose their works, as well as RM detecting the failures of NM and AM and recovering them. In addition, in Apache Mesos, the failure of the master is detected and automatically recovered, where the running tasks can continue to execute in the case of failover (Lynn, 2016). Accordingly, the tasks of a workflow will not be affected by the failure of RM (with work-preserving RM restart) in Apache YARN or the failure of the master in Apache Mesos. However, the responsibility of handling the failures of containers in Apache YARN is by frameworks themselves (Vavilapalli et al., 2013) as well as the failures of node and executor in Apache Mesos being reported to framework schedulers and letting them take the appropriate

[1] http://mesos.apache.org/documentation/latest/architecture/

actions to react to these failures, so that the responsibility of recovering from failures is by frameworks themselves. According to that, Apache YARN and Apache Mesos do not provide a mechanism to handle failures at application/framework level. As a result, Apache YARN and Apache Mesos use a reactive fault-tolerance approach for detecting and recovering from the failures of their masters, and have no mechanism for handling the failures at the level of workflow application and leave this responsibility to the workflow application itself, which reacts with its failures that may occur. For AWS Lambda as a serverless compute service, the underlying infrastructure is automatically managed, and in the cases where the Lambda function fails during processing an event, the functions invoked synchronously will reply with an exception and the functions invoked asynchronously are retried at least three times. If the input streams of Lambda function come from Amazon Kinesis streams and Amazon DynamoDB streams, these streams/events are retried until this function succeeds or the data expires, where the data remains in Amazon Kinesis streams and Amazon DynamoDB streams for at least 24 hours (Amazon, 2017b). Thus, it does not provide a mechanism to handle application-level failure, so that the fault-tolerance mechanism for workflow tasks is the responsibility of the workflow application.

2.7 Open Issues

In previous sections, several research studies have been highlighted that addressed big data workflow challenges and issues. Despite these efforts, some challenges are still open and not yet resolved, and others have not yet been investigated. In this section, we discuss key open research issues for big data workflow orchestration.

1. *Workflow Interoperability and Openness* – Since the execution of big data workflow is carried out in the cloud, there is an opportunity to achieve the level of interoperability between cloud-based workflow systems via standard models for interoperability and cooperation. Thus, the integrated execution of big data workflows from heterogeneous workflow systems and different cloud platforms is needed. It allows workflow reuse and automation, enables workflow sharing, and workflow migration.

2. *Workflow Fault-Tolerance and Dependability* – Several techniques and mechanisms of workflow fault-tolerance have been proposed to handle failures occurring during workflow execution and ensure its availability and reliability, but still supporting dependable big data workflow is a complex task. The dynamism of such workflows and execution environments as well as the lengthy execution process are all factors that need to be considered. Generally, handling the failures that occurred requires first catching the error, identifying its source, then reducing its impact and finally taking the appropriate actions to recover from it. Considering a "Cloud of clouds" environment, achieving those tasks is even harder due not only to the characteristics of big data and big data workflow, but also because of the characteristics of such environments.

3. *Distributed Workflow Execution* − The dynamism of big data workflow due to data coming in different formats, velocity and volumes (Zhou and Garg, 2015), poses the need for distributed execution of such workflow over clouds to gain the benefits of both parallel data processing and the dynamic nature of the execution environment, achieving data processing efficiencies and better performance. The Multicloud or "Cloud of clouds" architecture as an execution environment relies on multiple clouds makes such distributed execution possible. However, such architecture allows avoiding vendor look-in and provides more flexibility on the one hand, and on the other hand, it complicates the whole execution process and related processes such as scheduling and parallelisation, resulting in y challenges and issues still being open, such as balancing workloads among clouds or reducing the cost of moving large datasets between workflow tasks/fragments.

4. *Workflow Security* − Despite the benefits gained from using cloud computing and big data processing platforms, establishing standardised holistic solutions to security and privacy issues associated with moving big data workflow applications and their data to the cloud are still an important open issue. Comprehensive security solutions need to integrate the security of data-intensive tasks involved in workflow applications with the security of the consumed, generated and produced big data. The industry is further challenged by the regulatory requirements that are different in each jurisdiction, with a trend to become increasingly protective and prescriptive, as in the general data protection regulation in the European Union (regulation 'EU 2016/679'). Novel technologies, such as blockchain, provide potential solutions for trusted cloud provision of computational services, but at the same time pose new challenges with respect to privacy and scalability. Although many point solutions exist for security, and trusted platforms have been proposed (at operating system as well as application integration level), the above-mentioned increasingly challenging environment presents a need to expand on this through research in new security and privacy platforms for the ultra-dynamic environment of emerging big data workflows. Solutions may often not be technological only, but marry economic, business or personal incentives of stakeholders with the opportunities provided by technologies (see (Dong et al., 2017) for an example), thus providing solutions that are not only technically feasible but also leverage and align with stakeholder interests.

5. *User Perspective* − Despite the necessity of achieving the requirements of orchestrating big data workflow in the cloud, the requirements of users for the workflow should also be considered and accomplished. Thus, various requirements and constraints from different users result in different steps of a workflow needing to be executed, where the execution of these different steps might not be straightforward as the requirements may be conflicted. To clarify that, let us consider a data pipeline example in transportation, which is a workflow for analysing traffic flow on the roads. The driver and traffic red light management are examples of users for this workflow and these users define performance requirement as SLA requirements but from different contexts. For the driver, it would

be getting the analysed results for congestion on the road quickly, allowing him/her to slow down the car speed before this congestion; and for traffic red light management, it would be getting the analytical results on the density of roads and traffic volume changes quickly, allowing it to react accordingly to avoid any congestion that could happen.

6. *Cross-Cloud Workflow Migration Management* − The important open research issues relating to workflow migration management are: (1) finding the equivalent instances in the target cloud environment since the exact equivalent for instances between original cloud and target cloud may not exist, and (2) transferring large datasets to the target cloud environment in the case where such data is stored in the original cloud. These issues complicate the workflow migration task. And along with the absence of universally accepted standards that make the communication with the cloud uniform, and provisioning and managing cloud resources (Kashlev and Lu, 2014), it poses the need to deal with vendor-specific platforms at the target cloud.

7. *Workflow Resources Operability and Volatility* − With different clouds, creating and registering virtual machine images for cloud resources differs. The open challenge here is selecting or customising images offered by cloud providers in order to achieve different requirements of orchestrating big data workflow in the cloud. For example, different tasks of a workflow may require different software stacks to run, which means different images are required. Moreover, virtual resources may be provisioned from different clouds. As such maintaining and tracking these resources during the whole execution of big data workflow is a difficult issue since those resources are distributed and reside in various cloud platforms, and are provisioned and released on demand.

   Although cloud computing provides cloud resources on demand, the dynamic nature of cloud resources poses the need to deal with their volatilities because the loss of those resources often happens as a consequence of different failures (Kashlev and Lu, 2014). That is crucial for big data workflow since the execution of such workflow is usually a lengthy process. Therefore, the configuration of virtual machines required for running workflow tasks, the new data products attached to virtual machines and the intermediate and output big data products generated must all be stored and maintained during the whole execution of workflow. This is required to avoid any unexpected losses due to the loss of resources, whether they were virtual machines and/or storage volumes.

8. *Cross-Layer Resources Configuration Selection* − With different software-based data processing primitives (such as batch processing or stream processing) implemented by different PaaS-level big data programming frameworks on IaaS-level resources, there is a need for cross-layer resource configuration selection techniques. The open issue here is to automatically select the configurations for both IaaS-level resource and PaaS-level frameworks to consistently accomplish the anticipated workflow-level SLA requirements, while maximising the utilisation of cloud datacenter resources (Ranjan et al., 2015).

2.8 Summary

In this chapter, we outlined requirements for big data workflows in the cloud, presented a research taxonomy and reviewed the approaches and techniques available for executing big data workflow in the cloud. We also reviewed big data workflow systems and compared them against the presented research taxonomy. Further, we discussed research problems that are still not addressed. One of these research problems is distributed workflow execution. For stream workflows, as one of big data workflow applications, the lack of research supporting the distributed execution of this workflow leads to the problem of meeting real-time data processing requirements that are specified by the owner of such workflow. Therefore, we need to investigate the scheduling and execution of stream workflow applications over multiple cloud infrastructures to ensure efficiency and meet user-defined real-time performance requirements. This thesis carried out that investigation in steps due to the complexity and dynamism of stream workflow such as the distribution of data sources, data processing constraints, changing data velocity and structure of data pipeline at runtime. It first studies the behaviour of stream workflow and then investigates the static scheduling problem, and after that discusses the dynamic scheduling problem for different dynamic forms. The next chapter discusses the need for simulating the behaviour of stream workflows in Multicloud environment and proposes a new simulation toolkit named IoTSim-Stream.

Chapter 3

IoTSim-Stream: Modelling Stream Workflows in Cloud Simulation

Existing simulation toolkits do not support real-time processing or do not consider stream workflow that involves heterogeneous, complex and dynamic analytical components. Thus, given the current need of modelling and simulating the behaviour of stream workflows (aka stream graph applications) in a cloud computing environment, a new simulation toolkit is required. Therefore, in this chapter, we propose an IoT Simulator for big data stream processing (named IoTSim-Stream[1]) that offers a simulation environment to execute complex stream graph applications in a Multicloud environment, where large-scale simulation-based studies can be conducted to evaluate and analyse these applications. It also allows to evaluate the efficiency of new resource provisioning and scheduling policies in a repeatable, controllable and scalable environment.

[1]IoTSim-Stream has released as an open-source project and can be downloaded from: https://github.com/mutazb999/IoTSim-Stream

3.1 Introduction

Stream workflow application involves multiple streaming big data applications, where their execution should respect user performance requirements. Studying how stream workflow applications will perform in the cloud and evaluating the efficiency of new scheduling and resource allocation algorithms for such applications currently is not an easy task. These problems are often hard to investigate on real-world cloud infrastructures due to the following reasons: (1) unstable and dynamic nature of cloud resources, (2) scalability and complex requirements of stream workflow applications, and (3) real experiments on large, heterogeneous and distributed cloud platforms are subject to the impact of external events, are notably cost-ineffective, considerably time consuming and different conditions cannot easily reproduce results. The visible approach for evaluating application a benchmarking study in repeatable, controllable, dependable and scalable environments is via simulation toolkits, where experimental results can be reproduced easily (Calheiros et al., 2011). Therefore, a simulator supporting stream graph applications is a useful software toolkit, allowing both researchers and commercial organisations to simulate their stream graph applications and evaluate the performance of their algorithms in heterogeneous cloud infrastructures in an efficient time and with no cost.

To address the above research problems, we design and implement an IoT simulator for stream graph applications (IoTSim-Stream) that extends a popular and widely used cloud computing simulator (CloudSim), where we model stream workflow application in a Multicloud environment. It provides the ability to model and simulate the execution of stream graph application over resources provisioned from various cloud infrastructures.

This chapter is structured as follows: Section 3.2 describes what stream graph application is, while Section 3.3 outlines design issues for this type of workflow application. Section 3.4 reviews related simulation tools. Section 3.5 presents the architecture of the proposed simulator (IoTSim-Stream), while in Section 3.6, we explain in detail the implementation of IoTSim-Stream including the extended XML structure, proposed provisioning and scheduling policy, proposed stream scheduling policy and proposed VM-level scheduler. Section 3.7 presents our experiments to validate and evaluate the performance and scalability of IoTSim-Stream in simulating stream graph applications in Multicloud environment, and discusses the obtained results. Section 3.9 concludes the chapter.

3.2 Stream Graph Application

A stream graph application is a network of streaming data analysis components, where each individual component can be considered as a service and is executed independently over compute resources that are provisioned from the cloud, even though data dependencies among services should be maintained. Figure 3.1 presents an example of a stream graph application with its data processing requirements. The execution of this type of workflow application is

continuous (i.e. not one-time execution). It starts when the data streams generated by external sources such as sensors being continuously injected into data pipeline (particularly as input data streams to services). The data processing on these input data streams is continuously carried-out by those services to produce continuous output data streams (i.e. online insights), which are results of data processing computations. These output data streams generated by internal sources (i.e. parent services) are continuously injected into data pipeline, specifically as input data streams to child services, which process them continuously and then inject the results of computations into data pipeline. Therefore, we simply can say that this graph application has three main characteristics: continuous input data streams from external sources towards connected services and from internal sources (as results of computations that routed from these internal sources (i.e. parent services) to child services), continuous data processing of input data streams and continuous output data streams that are results of data processing computations at graph services.

As noted in Figure 3.1, each service has data processing requirement (the number of instructions required to process one MB of data stream) and data processing rate (the amount of streaming data the service can process per second such as 30MB/s). The owner of stream graph application can define user specific performance constraints in term of data processing rates on services, where these constraints are always maintained during the execution of this application. In case of no user performance constraint is specified on the service or the value defined is less than the speed of incoming streams, the total size of incoming streams for this service will be considered as a performance constraint. During the continuous execution of stream graph application, each service receives streaming input data from external sources and/or internal sources (i.e. parent services), processes them continuously as they arrive and generates streaming output data as results of computations which routed towards one or more child services based on the specified data modes (replica or partition). With replica mode, the output stream of parent service is replicated on child service(s) while with partition mode, the output stream of parent service is partitioned into portions based on the pre-defined partition percentages and then each portion is routed to corresponding child service. The end service(s) produces streaming output results for the execution of this graph application.

3.3 Design Issues of Stream Graph Application

Unlike batch-oriented data processing model that intends to process static data (i.e. the amount of input data is finite and it is stored before being processed and analysed) (Hirzel et al., 2013) (Hu et al., 2014), stream-oriented data processing model is intended for processing continuous data to gain immediate analytical insights. With this model, data arrives in streams, which are assumed to be infinite and are being processed and analysed (in a parallel and distributed manner) as they arrive to produce incremental results at the earliest they are prepared (Hirzel et al., 2013) (Hu et al., 2014). Based on this model, stream applications have been developed to process continuous data to produce continuous analyt-

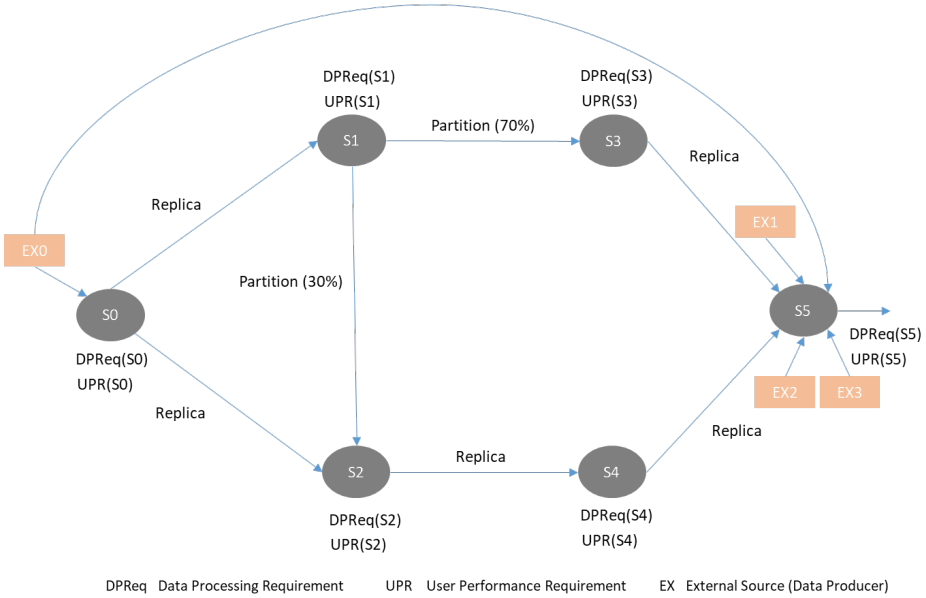


Figure 3.1: Sample stream graph application

ical results. However, given the demand of composing those applications into data pipelines forming stream graph application, this graph application has specific design issues. In the subsequent paragraphs, we will review these issues.

Modeling of graph nodes – Streaming data applications included in stream graph application can be considered as services since they can be separately running over any virtual resources, even though the data dependencies among them should be maintained. These independent processing nodes are allocated to appropriate VMs according to their performance requirements for processing continuous streams of data and producing analytical insights at the earliest they are prepared. Therefore, the simulator should models the nodes of graph as services adhering the data dependencies among them.

Modeling of data flows – The flow of data in this type of workflow application is streams, which are infinite continuous events. These streams are continuously injected as inputs into nodes (services) and continuously produced as results of computations, i.e. outputs of nodes (services). The simulator should thus represent this type of data as a sequence of events and allow transmitting them among VMs hosted by various datacentres.

Synchronisation of data flows – In stream graph application, there exists data flow dependencies across analytical nodes, resulting in the need of data flow synchronisation. Therefore, the execution of nodes (services) requires dynamic synchronisation of the states of parent and child services, e.g. output stream of parent service forms the basis of input data stream to one or more child services. Hence, the simulator should preserve the synchronisation of data flows as it directly impacts the correctness of stream graph application execution.

Modelling of Multicloud environment and its network performance – As the execution of stream graph application will be carried-out over multiple cloud infrastructures, the simulator should model Multicloud environment as an execution environment for this application. Not only this, but also the execution of this application on resources provisioned from multiple clouds means by the way that the streams of data are being transferred between VMs of datacentre (inbound traffic) or between different VMs hosted by various cloud datacentres (outbound traffic). Therefore, the simulator also requires to model the inbound and outbound network performance (i.e. bandwidth and latency) between cloud datacentres being used during simulation runtime, as the amount of streams being transferred is subject to the availability of bandwidth and the amount of delay.

3.4 Related Simulation Frameworks

With the emerging of cloud computing, various simulation-based toolkits have been developed in order to model the behaviour of different cloud services and applications on cloud infrastructures. These simulators help researchers in evaluating the performance of these systems and applications in controllable environment.

To the best of our knowledge, there is no simulator that model the execution of stream graph application in various resources provisioned from multiple cloud infrastructures. The most related simulators proposed by previous research works are described in the below paragraphs.

CloudSim (Calheiros et al., 2011) – It is a popular and widely used event-based simulator that models and simulates cloud computing infrastructures, applications and services. As an extensible and customisable tool, it allows to model custom cloud application services, cloud environments and application scheduling and provisioning techniques. In this simulator, users create cloud tasks (named Cloudlets) to define their workloads and then submit them to Virtual Machines (VMs) provisioned from cloud datacentre to be processed in the cloud. The application model of CloudSim is simpler and is more appropriate to simulate batch tasks, so that it is not capable to support stream tasks (i.e. continuous computation).

NetworkCloudSim (Garg and Buyya, 2011) – It is a simulation toolkit that models cloud datacentre network and generalises applications (e.g. High Performance Computing and e-commerce). It allows computational tasks involved in these applications to communicate with each other. NetworkCloudSim supports advanced application models and network model of datacentre, allowing researchers to accurately evaluate the new scheduling and provisioning techniques in order to enhance the performance of cloud infrastructure. Despite the advanced application models (i.e. multi-tier web application, workflow and MPI) supported by this simulator, the lack of application model that describes big data workflow applications is a major drawback in this simulator. Thus, it does not have the capability to

simulate stream tasks and even execute stream workflow applications in cloud environments.

MapReduce Simulators (MRPerf (Wang et al., 2009a), Mumak (Murthy, 2009) (Tang, 2009), SimMR (Verma et al., 2011), MRSim (Hammoud et al., 2010) and MR-CloudSim (Jung and Kim, 2012)) – MRPerf (Wang et al., 2009a) is phase-level simulator for MapReduce processing model. It serves as a design tool for analysing MapReduce based applications performance on specific configurations of Hadoop system, and as a planning tool for evaluating the proposed designs and topologies of cluster. Mumak (Murthy, 2009) (Tang, 2009) is an Apache discrete event simulator for MapReduce verification and debugging. It takes as input the job trace data from real experiment along with the definition of cluster and then feed them into simulator to simulate the execution of jobs in the defined virtual cluster with various scheduling policies. SimMR (Verma et al., 2011) is MapReduce based simulator developed in HP lab. It takes as input the execution traces derived from production workloads and then reply them to facilitate performance analysis and evaluating of new scheduling algorithms in MapReduce platforms. MRSim (Hammoud et al., 2010) is a discrete event simulation tool that extends SimJava, a Java discrete event engine to simulate various types of MapReduce-based applications and uses GridSim for network simulation. It offers functionalities for measuring the scalability of MapReduce applications and studying the effects of various Hadoop setup configurations on the behaviour of these applications. MR-CloudSim (Jung and Kim, 2012) is a simulator tool for modelling MapReduce-based applications in cloud computing environment. It extended the feature of CloudSim to implement bare bone structure of MapReduce on CloudSim, supporting data processing operations with this model. Thus, MR-CloudSim provides the ability for examining MapReduce model in a cloud-based datacentre. However, these simulators are only intended to support data processing operations with MapReduce model, thus they lack of support for modelling the streaming big data applications and even streaming big data workflow applications.

IoTSim (Zeng et al., 2017) – It is a software toolkit that built on top of CloudSim to simulate IoT applications in the cloud infrastructure. It integrates IoT application model to allow processing of IoT data by the use of big data processing platform in cloud infrastructure, providing both researchers and commercial entities with the ability to study the behaviour of those applications in controllable environment. This simulator is intended to support IoT application with MapReduce model, where it lacks the support for stream computing model. Therefore, it neither simulates stream big data application nor stream workflow application.

CEPSim (Higashino et al., 2016) – It is a simulator for event processing and stream processing systems in the cloud computing environment. It uses query model to represent user-defined query (application), where the modelled query (with all its vertices) is allocated to a VM to be simulated at once. With such simulator and by default, users have to determine manually the placements of their queries when submitting them to CEPSim. Therefore, the

main drawbacks of this simulator are (1) the user-defined query is executed entirely in a single VM, (2) provisioning resources according to input event streams of query is missing and (3) mapping of vertices to VMs is manual.

WorkflowSim (Chen and Deelman, 2012b) – It is a simulation toolkit that extends CloudSim to support scientific workflow scheduling and execution in the cloud with consideration of system overheads and failures. It incorporates model of workflow managements systems (similar to Pegasus workflow management system) in the cloud simulation environment, enabling researchers to study and evaluate the performance of workflow optimisation algorithms and methods more accurately. This simulator is intended to support scientific workflow applications, thus it lacks the support for big data workflow applications (batch, stream or hybrid). Therefore, it neither simulates streaming big data applications nor streaming big data workflow applications.

WRENCH (Casanova et al., 2018) – It is a simulation framework that extends SimGrid to provide high-level abstractions for simulating and executing workflow management systems. It allows studying the behaviour of workflow management systems in a simulation environment that is accurate, scalable and expressive. There are two categories of users for a Wrench. The first category includes users who are participating in research and development activities to create a simulation version of their workflow management system. The other category includes users who execute the simulated workflow management systems in order to study their behaviours with particular scientific workflows on particular simulated platforms. This simulator is intended to support scientific workflows, thus it neither simulates streaming big data applications nor streaming big data workflow applications.

Additionally, the common/shared drawback with all comparable simulators mentioned above is that they do not leverage the advantages of Multicloud environment to execute the modelled application on resources provisioned from various cloud infrastructures, where the proposed simulator supports that. This will open the door for further research studies including proposing resource and scheduling policies, improving performance and minimising execution cost. The summary of the above mentioned simulators along with their strengths and weaknesses are provided in Table 3.1.

3.5 The Proposed Architecture of IoTSim-Stream

The CloudSim is a simulation framework that models and simulates cloud infrastructures and services (Calheiros et al., 2011). It has rich features that make it the best choice to be the core simulation engine for our proposed simulator to simulate the behaviour of stream graph applications and their execution in Multicloud environment. Figure 3.2 shows the layered architecture of CloudSim with the essential elements of IoTSim-Stream (shown by orange-outlined boxes). In the subsequent paragraphs, we will describe these layers.

Table 3.1: Summary of Related Simulators

Simulation	**Core Engine**	PL	**Strengths**	**Weaknesses**
CloudSim (Calheiros et al., 2011)	GridSim	Java	• Model IaaS Cloud and batch tasks (long-running tasks) • Pluggable VM and application scheduling policy • Support of federated cloud environment	• Lack of modelling application models that have communicating tasks (Garg and Buyya, 2011) • Limited network support (Garg and Buyya, 2011)
NetworkSim (Garg and Buyya, 2011)	CloudSim	Java	• Model parallel applications such as (multi-tier web application, workflow and MPI) • Full network support (network-packet level) • Customise type of switches (root, aggregate and edge switch)	• Lack of big data applications support • Lack of support for stream tasks and even stream workflow applications • No Multicloud support

continued ...

... continued

Simulation	**Core Engine**	PL	**Strengths**	**Weaknesses**
MRPerf (Wang et al., 2009a)	CloudSim	Mix of C++, Tcl and Python	• Model big data batch processing (MapReduce) • Capture behaviour of Hadoop cluster • Simulate the full network by relying on ns-2	• Limited application behaviour (job has simple map and reduce tasks (Verma et al., 2011)) • Lack of stream / real-time processing support
Mumak (Murthy, 2009) (Tang, 2009)	Discrete event simulator engine	Java	• Verify/debug Hadoop MapReduce framework • Perform no actual I/O or computations • Simulate behaviour of production cluster	• No modelling of shuffle/sort phase (Verma et al., 2011) • No simulating of cloud resources • No job dependency • No modelling of failure correlations (only task-level failures) • Lack of stream processing support
SimMR (Verma et al., 2011)	Discrete event simulator engine	Not available	• Simulate MapReduce applications • Replayable MapReduce workload • Pluggable scheduling policy	• Lack modelling of cloud • Lack of stream processing support

continued ...

... continued

Simulation	**Core Engine**	PL	**Strengths**	**Weaknesses**
MRSim (Hammoud et al., 2010)	SimJava and GridSim package	Java	• Simulate Hadoop environment • Model shared Multi-core CPUs, HDD, and network topology and traffic • Consider cluster configurations	• Limited network support • Inherited limitations of SimJava (such as no support to create new simulation entity at runtime) • Lack of stream processing support
MR-CloudSim (Jung and Kim, 2012)	CloudSim	Java	• Model MapReduce-based applications	• Single-state map and reduce computation (Zeng et al., 2017) • Limited network support (no network link modelling) (Zeng et al., 2017) • No support to allow multiple MapReduce-based applications (Zeng et al., 2017) • Lack of stream processing support

continued ...

... continued

Simulation	**Core Engine**	PL	**Strengths**	**Weaknesses**
IoTSim (Zeng et al., 2017)	CloudSim	Java	• Model IoT application with MapReduce model • Allow to simulate multiple IoT applications • Model network and storage delays incurred during the execution of IoT-based applications	• Lack of stream processing support • Lack of big data workflows support
CEPSim (Higashino et al., 2016)	CloudSim	Scala and Java	• Model event processing queries • Customise operator placement, scheduling and load shedding strategies	• Limited network support • Query is executed entirely in a single VM • Provisioning resources according to input event streams of query is missing • Manual mapping of vertices to VMs
WorkflowSim (Chen and Deelman, 2012b)	CloudSim	Java	• Model scientific workflows • Consider diverse system overheads and failures	• No simulation of stream workflow applications

continued ...

...continued

Simulation	**Core Engine**	PL	**Strengths**	**Weaknesses**
WRENCH (Casanova et al., 2018)	SimGrid	C++	• Model scientific workflows • Implement simulation version of workflow management system • Execute simulated workflow management system	• No simulation of stream workflow applications

CloudSim Core Simulation Engine Layer This layer takes care of the interaction among the entities and components of CloudSim via message passing operations (Garg and Buyya, 2011). It offers numerous key functions, e.g. events queuing and handling, cloud entities creation (such as datacenter, broker), entities communication, and simulation clock management (Zeng et al., 2017). Entity within the ambit of the CloudSim is a component instance, which could be either a class or group of classes that depicts one CloudSim model (datacenter, broker) (Calheiros et al., 2011). It individually and independently exists, and has the capability for sending and receiving events to and from other CloudSim entities as well as process the received ones (Zeng et al., 2017). Event is a simulation event or message that passes among the CloudSim entities and holds relevant information, e.g. the type of event, time at which this event occurs as well as the data passed in this event to destination entity (Zeng et al., 2017).

CloudSim Simulation Layer This layer is designed to model the core elements of cloud computing. It contains several sub-layers to achieve that. The Network sub-layer models the topology of network among various datacentres, while Cloud Resource sub-layer models datacentre and cloud coordinator, thereby these components of those sub-layers allow to design IaaS environments (Zeng et al., 2017). The Cloud and VM Services sub-layers offer the functionality required for designing VM management and scheduling algorithms for cloud applications (Zeng et al., 2017). The sub-layer above, User Interface Structures, allows users to implement their structures for VM, cloud application and application cloudlet.

Service Layer This layer concentrates on orchestrating the execution of streaming data applications included in stream graph application.

User Code Layer This layer consists of two sub-layers, Scheduling Policy and Simulation Specification, providing the ability for users to specify their simulation configurations and scenarios in order to validate their scheduling and provisioning algorithms (Garg and Buyya, 2011).

The descriptions of IoTSim-Stream elements are as follows:

- Graph Application − It is a Directed Acyclic Graph (DAG) that represents a graph application.
- Graph Application Configuration − It defines simulation runtime, application and user requirements.
- Graph Application Engine (GraphAppEngine) − It parses DAG input file and handles the whole execution process of graph application. This process includes provisioning VMs from different providers, scheduling services of graph application on the provisioned VMs and the submission of graph application cloudlets to those VMs.

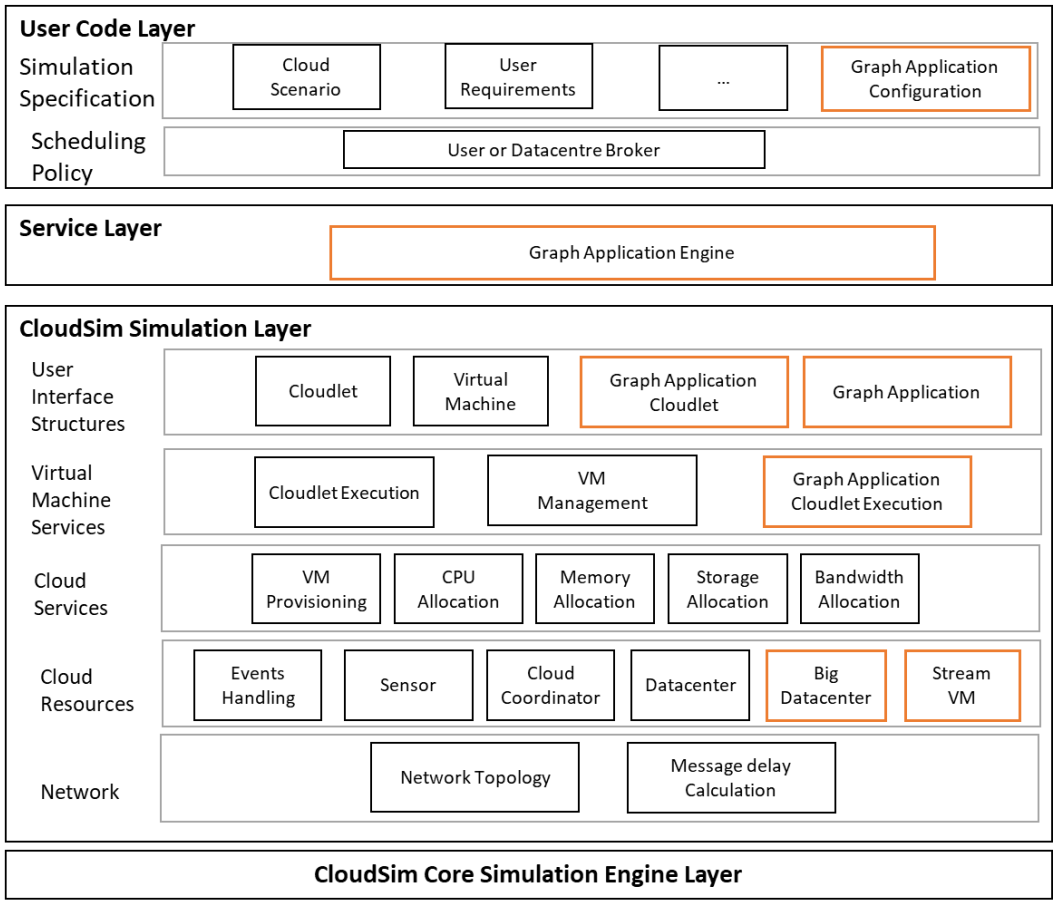


Figure 3.2: The proposed architecture of IoTSim-Stream (CloudSim with IoTSim-Stream elements)

- Graph Application Cloudlet (GraphAppClouldlet) – It represents a graph application with multiple stream application nodes (i.e. services).
- ServiceCloudlet – It represents a generalised stream application node..
- Graph Application Cloudlet Execution – It executes the submitted cloudlet (i.e. ServiceCloudlet) on VM.
- Big Datacenter (BigDatacenter) – It represents a cloud resource whose has a list of virtualised hosts, offers various flavours of VM, where the provisioned VMs are allocated on these hosts.
- Stream VM (SVM) – It represents a cloud resource where the mapped ServiceCloudlet will be executed on it.

3.6 Implementation

As we mentioned before, the proposed simulator (IoTSim-Stream) extends CloudSim with new functionality to support modelling the execution of stream graph application in multiple

cloud infrastructures. In line with the aforesaid design issues and requirements, the implementation of this simulator consists of two parts, which are modification and addition. The modification part is to modify the original code of CloudSim components such as datacenter and VM. While, the addition part is to add more components to meet the new requirements such as GraphAppEngine.

Figure 3.3 shows the class diagram of IoTSim-Stream. The components with orange-outlined boxes as shown in this figure can be classified either into an entity or a class as follows:

- Main entities
  - GraphAppEngine: It extends SimEntity to handle the execution of stream graph application. That is including workflow provisioning and scheduling, Data Producers (DPs) starting-up and shutting-down, and simulation shutting-down based on pre-defined simulation time.
  - BigDatacenter: It extends native Datacenter, which is an SimEntity, to support simulation of stream graph application that includes handling of VMs and transferring streams in between VMs and out of this datacentre to other datacentres.
  - External Source: It extends SimEntity to represent any kind of DP connected to the data source such as sensor, device or application and generates a continuous data stream.
- Classes for modelling Multicloud environment
  - VMOffers: It is an abstract class that encapsulates VM instance options offered by different cloud service providers such as Microsoft Azure, Amazon EC2 and Google Compute Engine. Each implementation of this abstract class represents the VM options offered by a particular cloud provider.
  - VMOffersBigDatacenter: It extends VMOffers abstract class to encapsulate different VM options offered by a particular cloud provider (i.e. a BigDatacenter).
  - SVM: It is an extended class of the core VM object to model a VM with input and output stream queues, to be a Stream VM.
  - ProvisionedSVM: It is a class designed to encapsulate a provisioned SVM with its information including the start and end time, and the cost.
- Classes for modelling basic BigDatacenter network
  - Channel: It is a class designed to represent a channel, which can be either ingress channel for transmitting streams between SVMs located at the same datacentre or egress channel for transmitting streams among SVMs located at different datacentres. It controls the amount of data transmitted in a shared data medium.

Each channel, whether ingress or egress, is a shared channel among different simultaneous stream transmissions (time-shared mode).

- StreamTransmission: It is a class represents transmission of a stream from source SVM to destination SVM located at the same datacentre or at different datacentres.

- Classes for modelling stream graph application

  - GraphAppCloudlet: It is a class designed to represent a stream graph application with multiple graph nodes i.e. services as described in the XML file of this graph application.
  - Service: It is a class designed to model atomic node in stream graph application as a service that processes incoming data stream(s) and produce output stream. It contains service information including service identification, data processing requirement, user performance requirement, its ServiceCloudlets, dependencies streams, parent service(s), child service(s) and output stream.
  - ServiceCloudlet: It is an extended class of the core Cloudlet object to implement an atomic graph node, which will be submitted to the cloud datacentre (i.e. BigDatacenter) by GraphAppEngine and executed in SVM. The atomic graph node or service can be modelled using one or more ServiceCloudlets. That is allowing parallel execution of service computations, and enhancing scalability and overall execution performance while meeting user performance requirements easily. Of course, each ServiceCloudlet contains the information of service to which it belongs.
  - Stream: It is a class designed to model data unit that being processed in this simulator. This class is used to represent both stream and stream portion when the original stream splits into several portions.

- Classes for scheduling ServiceCloudlets

  - Policy: It is an abstract class that implements the abstract policy for provisioning resources and scheduling of stream graph application (represented in DAG) in an IaaS datacentre. This class performs common tasks such as parsing the XML file describing the DAG, printing the scheduling plan, and returning provisioning and scheduling decisions to the GraphAppEngine.
  - SimpleSchedulingPolicy: It is an extended class from policy abstract class that represents the implementation of simple provisioning and scheduling policy for stream graph applications. It is first responsible for selecting the most suitable SVMs for each service whose achieved user performance requirement, and then scheduling the ServiceCloudlets of this service on them for execution. The detailed

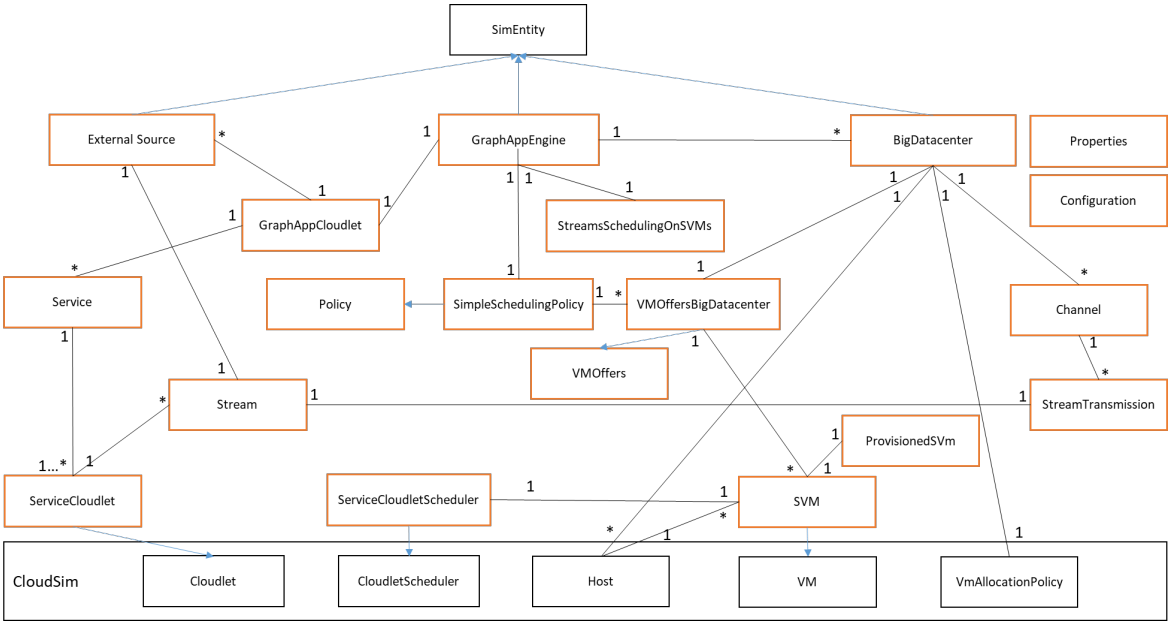


Figure 3.3: Class diagram of IoTSim-Stream

of this scheduling policy that offered in our simulator will be discussed in the next section.

- ServiceCloudletScheduler: It is an extended class of the core CloudletScheduler object to implement a space shared scheduling policy performed by SVM to run ServiceCloudlet. The detailed of this scheduler will be discussed in the next section.

- Class for scheduling streams on SVMs
  - StreamSchedulingOnSVMs: It is a class designed to schedule the divided portions of each stream either input or output stream on SVMs of destination service according to their computing powers.

- Classes for customising simulation parameters
  - Properties: It is an enumeration class represented the customisable parameters from simulation that are defined in simulation properties file (named simulation.properties).
  - Configuration: It is a class implements properties manager, which loads simulation properties file (i.e. simulation.properties) that contains parameters of simulation that are customised by users.

3.6.1 Extending XML Structure of Synthetic Workflows

Common workflow structures from different application domains (Bharathi et al., 2008), such as Montage in Astronomy, Inspiral in Astrophysics, Epigenomics in Bioinformatics and

CyberShake in Earthquake science, operate on static data inputs and produce final outputs. Thus, the XML structure generated by a set of synthetic workflow generators is described these static workflows and its parameters. However, the use of these workflow structures to simulate stream graph applications is practically feasible, as each job is considered as a service and the data flow becomes streams of data. The inputs of a job incoming from static files (not from parent jobs) become the continuous inputs of a service from DPs (i.e. external sources). The service continuously processes incoming data streams and continuously produces output stream. The output of a parent job, which is sent to one or more child jobs, becomes the continuous output of a parent service that is sent to one or more child services.

Accordingly, there is a need to extend the original structure of those workflows to describe the additional parameters and attributes of stream graph applications such as data processing requirements, input and output data rates. By making this extension, those workflow structures become stream graph structures. Table 3.2 lists the parameters and attributes being used in the extended XML structure.

Table 3.2: Additional Parameters of Stream graph Application

Parameter	**XML Attribute Name**	**Data Type**	**Value**
Data Processing Requirement	dataprocessingreq	Integer	ex. 1000 (in MI/MB)
User Performance Requirement	userreq	Number	ex. 10 (in MB/s)
Reference Input from Parent Service	serviceref	String	Referenced id of parent service as defined in XML file (ex. ID00001)
Processing Type of Input from Parent Service	processingtype	String	replica or partition
Partition Processing Type for Input Stream	partitionprecentage	Integer	1 - 99
External Source Identifier	id	String	ex. PID00000
External Source Name	name	String	ex. Producer0
External Source Data Rate	datarate	Number	ex. 12.5 (in MB/s)
Reference Input from External Source	producerref	String	The referenced id of external source as defined in XML file (ex. PID00000)
Service Output Data Rate	size	Number	ex. 20 (MB/s)

To aid understanding how to describe stream graph application in the extended XML structure using the presented parameters and attributes in Table 3.2, we use the presented sample stream graph application in Figure 3.1 and depict its XML structure in Listing 3.1.

Listing 3.1: Extended XML structure of sample stream graph application

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- generated: 2018-02-27:11:00 -->
<!-- generated by: Mutaz -->
<adag xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0" count="6" name="
    SampleStreamGraphhApplication" serviceCount="6" childCount="5">
<!-- part 1: list of all referenced outputs of services (may be empty) -->
<!-- part 2: definition of all services (at least one) -->
<externalsources>
<exsource id="PID00000" name="Producer0" type="stream" datarate="10"/>
<exsource id="PID00001" name="Producer1" type="stream" datarate="10"/>
<exsource id="PID00002" name="Producer2" type="stream" datarate="5"/>
<exsource id="PID00003" name="Producer3" type="stream" datarate="5"/>
<exsource id="PID00004" name="Producer4" type="stream" datarate="5"/>
</externalsources>
<service id="ID00000" dataprocessingreq="400" userreq="10" namespace="Sample"  name="BigService0" version="1.0">
<uses link="input" type="stream" producerref="PID00000"/>
<uses link="output" type="stream" size="5"/>
</service>
<service id="ID00001" dataprocessingreq="1000" userreq="5" namespace="Sample" name="BigService1" version="1.0">
<uses link="input" type="stream" processingtype="replica" serviceref="ID00000"/>
<uses link="output" type="stream" size="10"/>
</service>
<service id="ID00002" dataprocessingreq="500" userreq="8" namespace="Sample" name="BigService2" version="1.0">
<uses link="input" type="stream" processingtype="replica" serviceref="ID00000"/>
<uses link="input" type="stream" processingtype="partition" partitionprecentage="30" serviceref="ID00001"/>
<uses link="output" type="stream" size="8"/>
</service>
<service id="ID00003" dataprocessingreq="2000" userreq="7" namespace="Sample" name="BigService3" version="1.0">
<uses link="input" type="stream" processingtype="partition" partitionprecentage="70" serviceref="ID00001"/>
<uses link="output" type="stream" size="1"/>
</service>
<service id="ID00004" dataprocessingreq="3000" userreq="8" namespace="Sample" name="BigService4" version="1.0">
<uses link="input" type="stream" processingtype="replica" serviceref="ID00002"/>
<uses link="output" type="stream" size="2"/>
</service>
<service id="ID00005" dataprocessingreq="1500" userreq="38" namespace="Sample" name="BigService5" version="1.0">
<uses link="input" type="stream" producerref="PID00000"/>
<uses link="input" type="stream" producerref="PID00001"/>
<uses link="input" type="stream" producerref="PID00002"/>
<uses link="input" type="stream" producerref="PID00003"/>
<uses link="input" type="stream" producerref="PID00004"/>
<uses link="input" type="stream" processingtype="replica" serviceref="ID00003"/>
<uses link="input" type="stream" processingtype="replica" serviceref="ID00004"/>
<uses link="output" type="stream" size="4"/>
</service>
<!-- part 3: list of control-flow dependencies (may be empty) -->
<child ref="ID00001">
<parent ref="ID00000"/>
</child>
<child ref="ID00002">
<parent ref="ID00000"/>
<parent ref="ID00001"/>
</child>
<child ref="ID00003">
<parent ref="ID00001"/>
</child>
<child ref="ID00004">
<parent ref="ID00002"/>
</child>
<child ref="ID00005">
<parent ref="ID00003"/>
<parent ref="ID00004"/>
</child>
</adag>
```

3.6.2 Stream Scheduling

Since achieving user-defined performance requirement for a service may need more than one SVMs, this service will need more than one ServiceCloudlets, where each one is mapped to one SVM, leading to this service being mapped to more than one SVMs. Therefore, the

incoming data streams from external sources and parent services toward this service should be divided into portions and distributed across its SVMs according to their computing powers. Similarly, the output data stream producing by parent service towards child service(s) should be divided into portions and sent to the SVM(s) provisioned for such child service.

Consequently, we implement stream scheduling policy defined in the StreamSchedulingOnSVMs Java class. This policy divides each data stream into portions and schedules them in round-robin fashion according to computing power of SVMs of destination service. For instance, if one of child services in stream graph application has two SVMs, where the computing power of first VM is twice computing power of the second one, the divided portions of one output stream of parent service are distributed into 2:1 way - two portions for first VM and one portion for second VM.

3.6.3 Scheduler and Execution of ServiceCloudlet

Before providing the details of the implemented scheduler in IoTSim-Stream, we need to discuss how IoTSim-Stream is initialising and what is the provisioning and scheduling policy being used to schedule stream graph application on Multicloud environment. Algorithm 1 shows the pseudo-code of simple provisioning and scheduling algorithm that we implemented in IoTSim-Stream. This algorithm provisions the most suitable VMs for services included in stream graph application which meet the user performance requirements for those services, where all VMs for a service are provisioned from one cloud-based datacentre. For each service, it finds VMs with higher computing powers upto the required MIPS value (that is calculated based on user performance requirement and service processing requirement) and provisions them to achieve as much as possible from this value. Then, it backs to VMs with lower computing powers to achieve the remaining value. Nevertheless, in case of the selected VMs list for any service is empty, IoTSim-Stream shows a message to the user indicating that, and then is terminated. This happens because there is no VM offer available in the selected datacentre that can achieve the required MIPS to process at least one stream unit according to the value of data processing requirement of such service. Therefore, the user in that case can either reduce the value of minimum stream unit (leading to reduction in the value of required MIPS for processing one stream unit) or add VM offer that satisfies processing at least one stream unit for this service.

Figure 3.4 presents the flow of communication for initialising IoTSim-Stream, provisioning SVMs and scheduling ServiceCloudlet on the provisioned SVMs. Once a stream graph application is submitted, GraphAppEngine handles this submission and sends to itself START_DELAY event to allow enough time for BigDatacenters to initialise. During processing this event, GraphAppEngine sends RESOURCE_CHARACTERISTICS event to each BigDatacenter and waiting for their replies. When all BigDatacenters send their replies as RESOURCE_CHARACTERISTICS events, GraphAppEngine processes them and then trigger the process of provisioning and scheduling such application by sending to it-

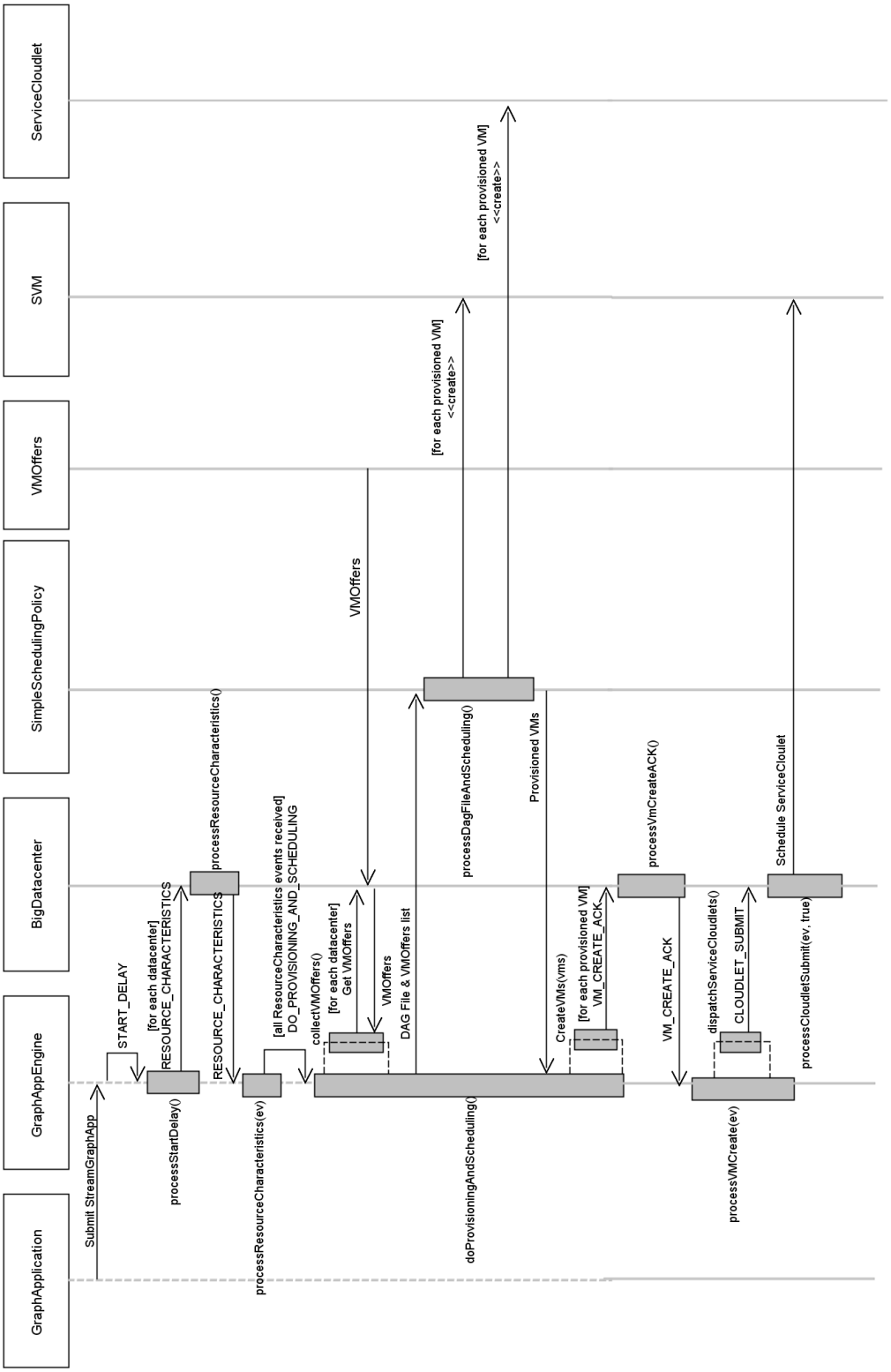


Figure 3.4: Sequence Diagram: The flow of communication for initialising IoTSim-Stream and scheduling ServiceCloudlet on SVMs

self DO_PROVISIONING_AND_SCHEDULING event. In doProvisioningAndScheduling() procedure, the following functions are performed:

1. Call collectVMOffers() procedure to collect all VM offers provided by BigDatacenters by querying them.

2. Send XML file of submitted application along with the list of VM offers to scheduling policy. This policy executes processDagFileAndScheduling() procedure to parse this file, extracts the structure of application, selects the best suitable SVMs and prepares the scheduling plan. After the selection of suitable VMs, the objects for SVM and ServiceCloudlet are created.

3. Retrieve the generated scheduling plan or table.

4. Use the generated scheduling plan to provision and create SVMs by sending messages (VM_CREATE_ACK) to corresponding BigDatacenters via event mechanism.

While receiving acknowledgements (i.e. VM_CREATE_ACK events) for SVM creations from BigDatacenttres, each acknowledgement for one SVM is processed as it arrives and the corresponding ServiceCloudlet is dispatched to this SVM by calling dispatchServiceCloudlets() procedure; this procedure sends CLOUDLET_SUBMIT event to corresponding BigDatacenter, which processes the received event (CLOUDLET_SUBMIT) and schedules this ServiceCloudlet on a SVM.

Figure 3.5 shows the process of sending data streams from external sources and transferring input and output data streams to and from SVMs. Once a stream graph application is being scheduled on SVMs (i.e. ServiceCloudlets of application services have been scheduled on SVMs and ready for execution), the GraphAppEngine sends to itself END_OF_SIMULATION event with the delay specified by user-defined requested simulation time; this event will being sent after this delay, which triggers the end of simulation process. Then, it sends SEND_STREAM events to all external sources requesting them to start sending their data streams to corresponding BigDatacenters, where these datacentres will forward those streams to respective SVMs. At that time, the simulation begins.

Each external source that receives SEND_STREAM event will process it and queries StreamSchedulingOnSVMs object about the portions of its stream and the information of BigDatacenters and SVMs where these portions should be transferred and available. When these portions are received along with the relevant information (i.e. destination BigDatacenters and SVMs), this external source immediately sends them as EXSOURCE_STREAM events to destination BigDatacenters. Each EXSOURCE_STREAM event will be processed by corresponding BigDatacenter whose will send to itself STREAM_AVAILABLE event. It then processes this event to make stream portion available in the corresponding SVM by adding such portion to the input queue of corresponding SVM and sends to itself a message (i.e. VM_DATACENTER_EVENT).

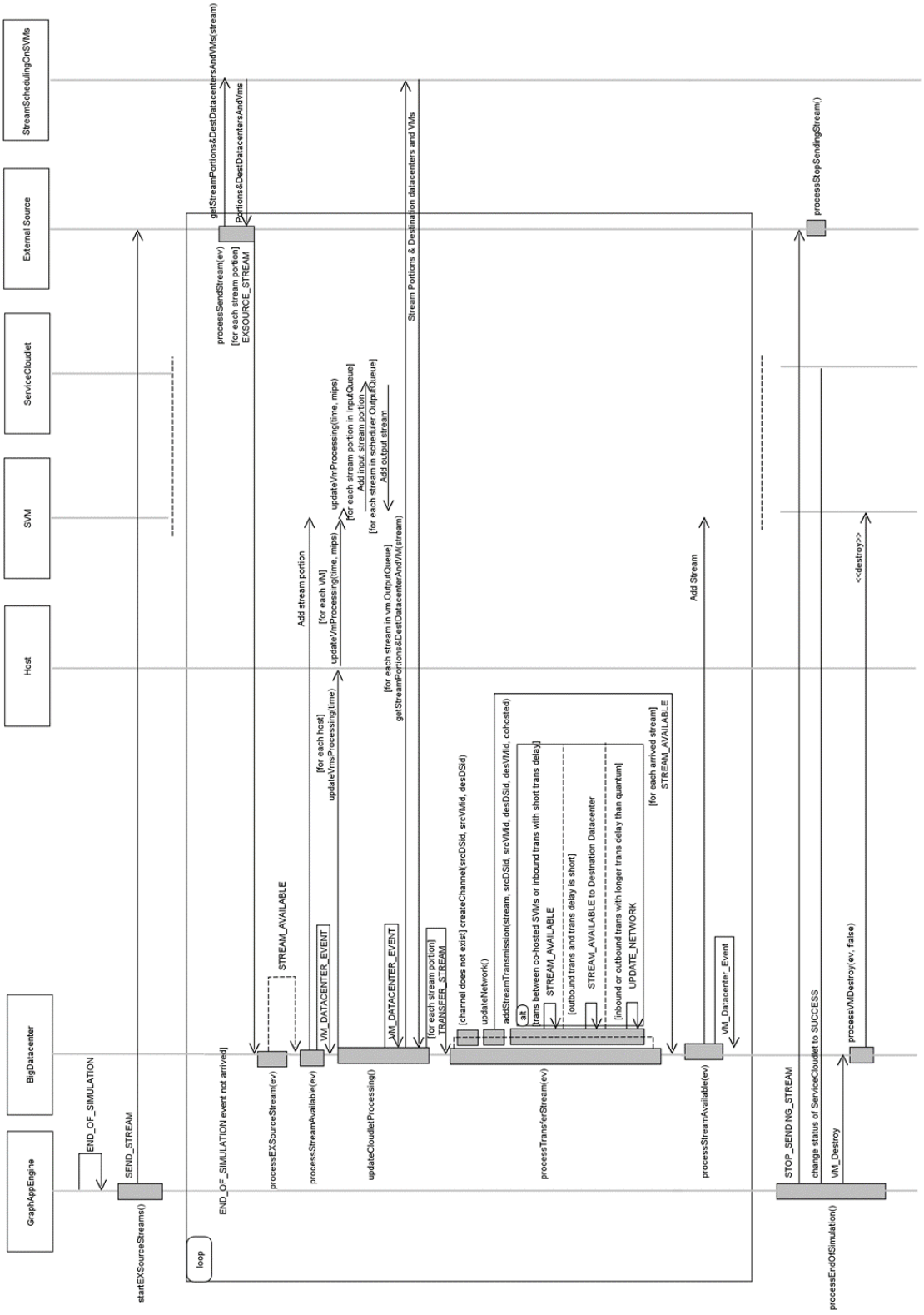


Figure 3.5: Sequence Diagram: Transferring and exchanging data streams among SVMs

Algorithm 1 Simple Provisioning and Scheduling Policy

Require: minDPUnit a data processing rate for minimum stream unit in an application

```
for each service in Services do
    selectedVMs ← φ
    requiredMIPS ← service.userreq * service.dataprocessingreq
    placementCloud ← pick random cloud from avaliable clouds
    VMOffers ← get VM flavours in accending order of power
    for each vm_i in VMOffers do
        vmMIPS ← get vm_i power
        if vmMIPS/service.dataprocessingreq < minDPUnit then
            continue
        end if
        if vmMIPS ≤ requiredMIPS then
            if i + 1 < n then
                nextvmMIPS ← get vm_{i+1} power
                if nextvmMIPS > requiredMIPS then
                    toProvisionVM ← true
                end if
            else
                selectedVMs ← selectedVMs ∪ vm_i
                requiredMIPS ← requiredMIPS − vm_i power
                i ← i − 1
            end if
        else
            if i − 1 ≥ 0 then
                previousVmMIPS ← get vm_{i−1} power
                if previousVmMIPS ≥ requiredMIPS &&
                        previousVmMIPS < vmMIPS &&
                    previousVmMIPS/service.dataprocessingreq ≥ minDPUnit then
                    i ← i − 2
                else
                    toProvisionVM ← true
                end if
            else
                toProvisionVM ← true
            end if
        end if
        if toProvisionVM == true then
            selectedVMs ← selectedVMs ∪ vm_i
            requiredMIPS ← requiredMIPS − vm_i power
            toProvisionVM ← false
        end if
        if requiredMIPS ≤ 0 then
            break
        end if
    end for
    if selectedVMs is empty then
        show message `provisioning failed' to the user
        terminate the currently running simulator (i.e exit)
    end if
end for
```

When VM_DATACENTER_EVENT being received by BigDatacenter, it processes this event and then updates the state of all simulated entities in a BigDatacenter. At this point, all stream portions available in input queues of all SVMs in all hosts will be moved to the input queues of corresponding ServiceCloudlets via their schedulers, making them available for processing. As well, all output streams available in output queues of ServiceCloudlets as results of computations will be moved to output queues of corresponding SVMs in order to be transferred later. Next, this BigDatacenter sends another VM_DATACENTER_EVENT to itself for future updating. After that, BigDatacenter starts the next communication flow to transfer output streams of ServiceCloudlets available at their SVMs to destination SVMs in order to be input streams for others ServiceCloudlets. Thus, BigDatacenter checks output queues of all hosted SVMs looking for any output stream available as a result of completed com-

putation. For each output stream available at a SVM, it queries StreamSchedulingOnSVMs object about the portions of such stream and the information of destination BigDatacenters and SVMs where these portions should be available. It then uses this information to send each stream portion to destination BigDatacenter as a message (i.e. TRANSFER_STREAM event) via event mechanism.

Each TRANSFER_STREAM event is processed by corresponding BigDatacenter whose creates an ingress or egress channel based on whether the transmission of included stream portion is inbound or outbound if such channel does not exist. It then updates its network and adds a new stream transmission to such channel for transferring stream portion. During the addition, if such transmission is between co-hosted SVMs or it is inbound transmission with short transmission delay, this BigDatacenter sends to itself STREAM_AVAILABLE event with cohosted delay for cohosted transmission or with ingress latency for inbound transmission. While if the transmission is outbound transmission and transmission delay is short, this BigDatacenter sends STREAM_AVAILABLE event to the destination BigDatacenter with egress latency to such BigDatacenter. Whereas in case of transmission delay for either inbound or outbound transmission is longer than the pre-defined minimum quantum of time between events (i.e. 0.01 second - 10ms), this BigDatacenter sends to itself UPDATE_NETWORK event with this delay. Furthermore in network update, this BigDatacenter updates the processing of stream transmissions in all ingress and egress channels, where for each arrived stream, it sends STREAM_AVAILABLE event with ingress or egress latency based on transmission type to corresponding destination BigDatacenter (i.e. itself or other BigDatacenter). Such Bigdatacenter will process this event to make the transferred stream portion available into the corresponding SVM.

Nevertheless, the whole process of transferring and exchanging streams among different SVMs hosted in different BigDatacenters continues until END_OF_SIMULATION event being received (i.e. thereafter the pre-defined delay at the begin of simulation). At that time, GraphAppEngine receives this event and processes it, and then starts the end of simulation process, which includes the followings:

1. Stop external sources from sending their streams to corresponding BigDatacenters by sending STOP_SENDING_STREAM events to them.

2. Change the status of all ServiceCloudlets to `Success', indicating the end of their executions.

3. Destroy all provisioned SVMs by sending VM_Destroy events to their BigDatacenters, which process these events and destroy the hosted SVMs.

When dealing with scheduling, CloudSim has two schedulers, which are VmScheduler and CloudletScheduler. The VmScheduler is host-level scheduler that can run either in space-shared or time-shared mode for allocating cores of processor from a host to VMs (i.e. virtual machine monitor allocation policy). While, the CloudletScheduler is VM-level scheduler that

can also run in one of the aforementioned modes for determining the computing power share between Cloudlets in a VM (Calheiros et al., 2011). Since each ServiceCloudlet is submitted to one SVM and this SVM needs to handle the continuous execution of this cloudlet to process incoming streams and produce output stream, the new VM-level scheduler is required. Therefore, we implement VM-level scheduler named ServiceCloudletScheduler for each SVM within IoTSim-Stream. Algorithm 2 shows the pseudo-code of ServiceCloudletScheduler. This scheduler runs in space-shared scheduling mode.

As the ServiceCloudletScheduler is running, it continuously checks its input queue (inputQueue) looking for any incoming streams. If inputQueue is not empty and the waitingStreamsForNextPC flag is true (see Line 35), the ServiceCloudletScheduler enters into the while-loop and performs the following steps on each iteration (see Line 37 - 55):

1. Fetches the head of inputQueue (i.e. input stream portion with least portion id) and dequeues this stream in case of this stream is not existing in working input stream list (workfingInputStream), and then adding it into such list and preparing for next PC (see Line 40 - 44).

2. Checks if all required stream portions for one PC arrive and they are added to workfingInputStream in order to perform the appropriate action (see Line 45 - 54):

    - If yes, it then checks if the time required to process streams included in this PC based on the capacity of cloudlet is less than 0.1, therefore it moves those streams to assumeProcessedStreams list and empties workfingInputStream list. Otherwise, it changes the flag of "check" to false. This flag helps to get out of while-loop either in case of stream portions included in workfingInputStream list need processing time at least as long as the minimum time for one PC (i.e. 0.1) or the head of inputQueue is stream portion for next PC based on portion id.
    - If no, it changes the "check" flag to false if the value of continueCheck flag is false. The continueCheck flag is used to continue in while-loop as the previous head of inputQueue has been dequeued from this queue (see Line 40 - 44), so that in next iteration, the next head can be fetched and checked to be either dequeued or not. That is very important to fetch and dequeue all of those streams required for one PC as they arrive and before the next update of the scheduler if possible, ensuring low-latency data processing.

When all required stream portions for one PC arrive and they are added to workfingInputStream, and waitingStreamsForNextPC flag is true, the scheduler calculates the total size of input stream portions and using it with the value of data processing requirement for a service to update the length of cloudlet. Then, it changes the startPC flag to true, which indicates the start of one PC and waitingStreamsForNextPC flag to false as we are in the phase of starting the execution of one PC (see Line 57 - 67). After that, the scheduler starts the execution of this PC to process the included stream portions in such PC and updates

completion/progress accordingly (see Line 10 - 13). While the execution of one PC and updating its progress, the scheduler also checks the completion of this PC, so that when the execution finishes (i.e. renaming cloudlet length equals zero), it performs the following steps (see Line 14 - 34):

1. Change the startPC flag to false, which indicates the end of execution of one PC (this PC).
2. Produce the output stream and add this stream into outputQueue.
3. Empty the working stream list (workfingInputStream)
4. Change the waitingStreamsForNextPC flag to true, which indicates the current status backs to wait for stream portions to be arrived if they are not arrived yet and to fetch those portions from input queue required to start new PC.

3.7 Validation and Evaluation

To validate and quantify the efficiency of IoTSim-Stream in simulating stream graph applications in Multicloud environment, we design two experiments, which are simulator validation, and performance and scalability evaluation. We conduct these experiments on a machine that had Intel Core i7-6600U 2.60GHz (with 2 cores and 4 logical processors), 16GB of RAM memory and running Windows 10 Enterprise, and then collecting the experimental results. In this section, we present our experimental methodology (including Multicloud environment configuration, network configuration, simulation configuration parameters and evaluation experiments) and discusses the experimental results.

3.7.1 Multicloud Environment

Multicloud environment consolidates multiple clouds in order to maximise the benefits from cloud services, which opens the door towards orchestrating the execution of multiple applications over various clouds. To model this environment for our experiments, we define two clouds (i.e. two cloud-based datacentres) and configure them as listed in Table 3.3. For each datacentre, we define four different flavours of VMs, which are Small, Medium, Large and Extra Large, where the configurations of VM vary from one datacentre to another, matching what the cloud datacentre is in real. Table 3.4 shows the configurations of VM for the both defined datacentres. This Multicloud environment configuration is consistent throughout the entire evaluation.

Algorithm 2 ServiceCloudletScheduler for scheduling and executing ServiceCloudlet on a VM

```
outputQueue ← φ
inputQueue ← φ {PriorityQueue sorting stream portions by ids - ascending order}
workingInputStreams ← φ {list of input streams for a Processing Cycle (PC)}
assumeProcessedStreams ← φ {list of input streams that is assumed to be processed}
startPC ← false {flag for starting one PC}
waitingStreamsForNextPC ← true
totalOutputSize ← 0
totalInputSize ← 0
for each ServiceCloudlet cl in CloudletExecList do {One ServiceCloudelt exists}
    if startPC == true then {when all required input stream portions are available for one PC}
        start processing stream portions in this cycle
        update the completion of this cycle
    end if
    if waitingStreamsForNextPC == false then {Execution of one PC for ServiceCloudlet is in progress}
        if rcl.getRemainingCloudletLength() == 0 then {Completion of one PC}
            startPC ← false
            produce output stream
            if totaOutputSize == 0 then
                totaOutputSize ←  size of output stream
            end if
            if totalInputSize == 0 then
                totalInputSize ← sum sizes of required input streams
            end if
            numOfProcessedStreams ← size of workingInputStreams + size of sassumeProcessedStreams size
            processedPortionsSize ← max portion size * numOfProcessedStreams
            proportionInToOut ← totalOutputSize/totalInputSize
            outputStreamSize ← processedPortionsSize ∗ proportionInToOut
            create output stream with outputStreamSize
            enqueue created output stream in outputQueue
            workingInputStreams ← φ
            assumeProcessedStreams ← φ
            waitingStreamsForNextPC ← true
        end if
    end if
    if inputQueue is not empty && waitingStreamsForNextPC == true then
        check ← true
        while check && inputQueue is not empty do
            continueCheck ← false
            stream_portion ← retrieve the head stream portion of this queue {not dequeue from priority queue}
            if stream_portion is not in workingInputStreams then
                stream_portion ← perofrm dequeue operation from inputQueue
                add stream_portion in workingInputStreams
                continueCheck ← true
            end if
            if stream portions required for one PC being arrived then
                if required MIPS for processing these streams in this PC / cloudlet capacity ¡ 0.1 then {0.1 is min.
                time for one PC}
                    move stream portions to assumeProcessedStreams list
                    workingInputStreams ← φ
                else
                    check ← false
                end if
            else if continueCheck == false then
                check ← false
            end if
        end while
    end if
    if stream portions required for one PC being arrived && waitingStreamsForNextPC == true then
        InPortionsSize ← sum sizes of input stream portions
        clLength ← service_processing_req ∗ InPortionsSize {length of ServiceCloudlet}
        if cloudlet length == 1 then {value assigned when cloudlet initialised}
            cl.length = ((current total length of ServiceCloudlet + clLength) / cpus) -1 {length in MIPS}
        else
            cl.length = (current total length of ServiceCloudlet + clLength) / cpus {length in MIPS}
        end if
        startPC ← true
        waitingStreamsForNextPC ← false
    end if
end for
```

Table 3.4: Types and Configuration of VMs in Modelled Datacenters

VM Type	**Datacenter 0**	**Datacenter 1**
Small	PEs: 2 MIPS: 1000 RAM (MB): 4096 Storage (MB): 8192 Bandwidth (MB/s): 1000	PEs: 2 MIPS: 2000 RAM (MB): 4096 Storage (MB): 8192 Bandwidth (MB/s): 1000
Medium	PEs: 4 MIPS: 1000 RAM (MB): 7168 Storage (MB): 16384 Bandwidth (MB/s): 1000	PEs: 4 MIPS: 2000 RAM (MB): 8192 Storage (MB): 18432 Bandwidth (MB/s): 1000
Large	PEs: 8 MIPS: 1000 RAM (MB): 14336 Storage (MB): 32768 Bandwidth (MB/s): 1000	PEs: 8 MIPS: 2000 RAM (MB): 16384 Storage (MB): 34816 Bandwidth (MB/s): 1000
Extra Large	PEs: 16 MIPS: 1000 RAM (MB): 30720 Storage (MB): 65536 Bandwidth (MB/s): 1000	PEs: 16 MIPS: 2000 RAM (MB): 32768 Storage (MB): 69632 Bandwidth (MB/s): 1000

Table 3.3: Configuration of Datacenters

Parameter Configuration	**Datacenter 0**	**Datacenter 1**
Hosts	1000	1000
PEs	64	64
MIPS per PE	1000	2000
RAM per Host (MB)	144000	176000
Storage per Host (MB)	1400000	1500000
VM Boot Delay Time (sec)	20	20

3.7.2 Network Configuration

Network performance of Multicloud environment determines the amount of data being transferred within cloud-based datacentre (ingress traffic) and between different cloud-based datacentres (egress traffic). For our experiments, we have conducted TCP bandwidth and latency tests between different zones of Nectar Cloud [2] (Nec, 2017) using IPerf [3] and Ping utility, and then collected the results for both bandwidth (in MB/s) and latency (in second). We chosen average values to model network performance for both ingress and egress traffic for cloud-based datacentres in the modelled Multicloud environment as listed in Table 3.5. Since studying the network performance is out of our scope and for simplicity purpose, we made the configuration of network performance for both cloud-based datacentres in the modelled environment is identical with slight difference. This configuration of network performance

[2]The National eResearch Collaboration Tools and Resources project (Nectar) provides cloud computing infrastructure for Australia's research community.

[3]IPerf is a cross-platform network performance measurement tool for both Transmission Control Protocol (TCP) and User Datagram Protocol (UDP)

Table 3.5: Configuration of Network Performance of Modelled Multicloud Environment

Network Parameter	**Datacenter 0**	**Datacenter 1**
Ingress Bandwidth	770 MB/s	780 MB/s
Ingress Latency	0.00077 second	0.00075 second
Egress Bandwidth	170 MB/s	180 MB/s
Egress Latency	0.028 second	0.026 second

Table 3.6: User-defined Simulation Parameters Configuration

Parameter	**Description**	**Value**
simulation time	The requested simulation time in seconds	Refer to experiments
scheduling.policy	Provisioning and scheduling policy	*SimpleSchedulingPolicy*
dag.file	Path of XML file of stream graph application	*path_value*
cloud.datacenter	Number of clouds in Multicloud environment, where each cloud is represented by a datacenter	2
engine.network.bandwidth	Network bandwidth of GraphAppEngine	300
engine.network.latency	Network latency of GraphAppEngine	0.05
cloud.provider	Index of cloud provider in Multicloud environment (index starting from 0)	*value*
datacenter.hosts#*index* (ex. datacenter.hosts#0)	number of hosts in datacenter	*value*
vm.delay#index	Average delay of VM boot time	*value*
vm.offers#index	Path of Java class for offerings of cloud-based datacentre	*packagename.classname*
host.cores#index	Number of cores (PEs) available for each host	*value*
host.memory#index	Amount of memory available for each host	*value (unit: MB)*
host.storage#index	Amount of storage available for each host	*value (unit: MB)*
core.mips#index	MIPS for each core or PE	*value*
internal.bandwidth#index	Internal network bandwidth available for each VM within cloud-based datacentre	*value (unit: MB/s)*
internal.latency#index	Network delay between VMs within cloud-based datacentre	*value (unit: MB/s)*
external.bandwidth#index	External network bandwidth available by cloud-based datacentre for transferring data streams to other datacentres	*value (unit: MB/s)*
external.latency#index	Network delay from cloud-based datacentre to other dataentres	*value (unit: MB/s)*

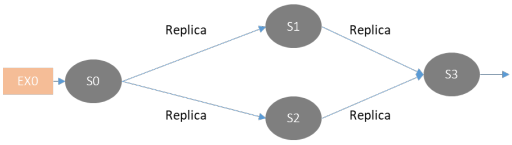
for those datacentres is consistent throughout the entire evaluation.

3.7.3 Simulation Configuration Properties

Prior to run the simulator, we need to configuration its parameters that are defined in simulation properties file (simulation.properties). These parameters will be read by IoTSim-Stream during initialisation for preparing to simulate given stream graph application according to the specified configurations. Table 3.6 shows the simulation parameters that included in this file with their description and values used in our experiments.

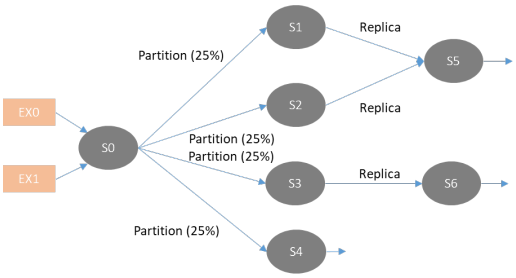
SERVICE / EXTERNAL SOURCE	DATA PROCESSING REQUIREMENT	USER PERFORMANCE REQUIREMENT	INPUT	OUTPUT	OUTPUT STREAM TO SERVICE(S)
EX0	N/A	N/A	N/A	4 MB/s	4 MB/s to S0
S0	500 MI/MB	4 MB/s	4 MB/s	4 MB/s	4 MB/s to S1
S1	500 MI/MB	4 MB/s	4 MB/s	4 MB/s	4 MB/s to S2
S2	1000 MI/MB	4 MB/s	4 MB/s	4 MB/s	4 MB/s to S3
S3	1000 MI/MB	4 MB/s	4 MB/s	1 MB/s	None

(a) Structure and configuration of stream graph application 1 (App1)



SERVICE / EXTERNAL SOURCE	DATA PROCESSING REQUIREMENT	USER PERFORMANCE REQUIREMENT	INPUT	OUTPUT	OUTPUT STREAM TO SERVICE(S)
EX0	N/A	N/A	N/A	4 MB/s	4 MB/s to S0
S0	500 MI/MB	4 MB/s	4 MB/s	4 MB/s	4 MB/s to S1 4 MB/s to S2
S1	500 MI/MB	4 MB/s	4 MB/s	4 MB/s	4 MB/s to S3
S2	1000 MI/MB	4 MB/s	4 MB/s	4 MB/s	4 MB/s to S3
S3	1000 MI/MB	8 MB/s	8 MB/s	1 MB/s	None

(b) Structure and configuration of stream graph application 2 (App2)



SERVICE / EXTERNAL SOURCE	DATA PROCESSING REQUIREMENT	USER PERFORMANCE REQUIREMENT	INPUT	OUTPUT	OUTPUT STREAM TO SERVICE(S)
EX0	N/A	N/A	N/A	2 MB/s	2 MB/s to S0
EX1	N/A	N/A	N/A	2 MB/s	2 MB/s to S0
S0	2000 MI/MB	4 MB/s	4 MB/s	16 MB/s	4 MB/s to S1 4 MB/s to S2 4 MB/s to S3 4 MB/s to S4
S1	1000 MI/MB	4 MB/s	4 MB/s	4 MB/s	4 MB/s to S5
S2	2000 MI/MB	4 MB/s	4 MB/s	4 MB/s	4 MB/s to S5
S3	1000 MI/MB	4 MB/s	4 MB/s	4 MB/s	4 MB/s to S6
S4	4000 MI/MB	4 MB/s	4 MB/s	1 MB/s	None
S5	4000 MI/MB	8 MB/s	8 MB/s	2 MB/s	None
S6	4000 MI/MB	4 MB/s	4 MB/s	1 MB/s	None

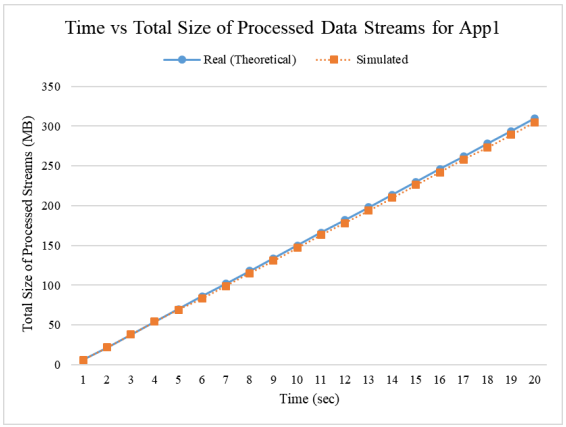
(c) Structure and configuration of stream graph application 3 (App3)

Figure 3.6: Stream graph applications with their parameter configurations for our experiments

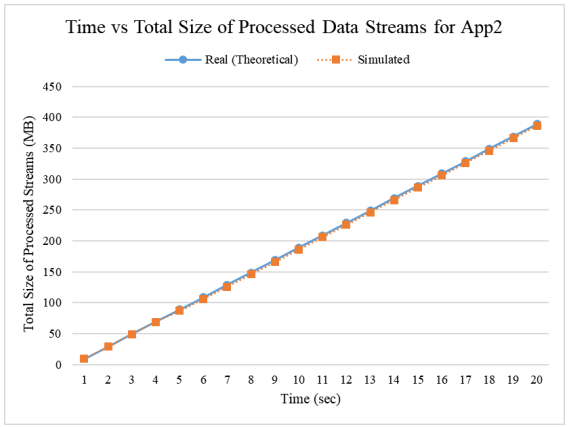
Table 3.7: Mapping Services of Stream Graph Applications on VMs

	App1	**App2**	**App3**
Required MIPS per Service	S0: 2000 S1: 2000 S2: 4000 S3: 4000	S0: 2000 S1: 2000 S2: 4000 S3: 8000	S0: 8000 S1: 4000 S2: 8000 S3: 4000 S4: 16000 S5: 32000 S6: 16000
VM Type per Service	S0: Small - Datacenter 0 S1: Small - Datacenter 0 S2: Small - Datacenter 1 S3: Small - Datacenter 1	S0: Small - Datacenter 0 S1: Small - Datacenter 0 S2: Small - Datacenter 1 S3: Medium - Datacenter 1	S0: Medium - Datacenter 1 S1: Medium - Datacenter 0 S2: Large - Datacenter 0 S3: Medium - Datacenter 0 S4: Large - Datacenter 1 S5: Extra Large - Datacenter 1 S6: Extra Large - Datacenter 0

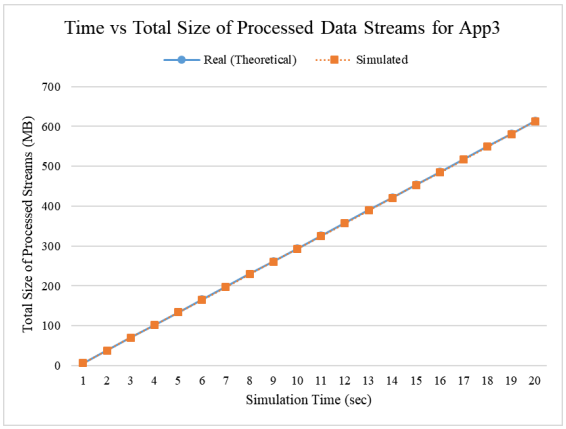
The parameters from "cloud.provider" to "external.latency" shown in Table 3.6 need to be repeated for each cloud provider (i.e cloud-based datacentre) defined in Multicloud environment. As we mentioned earlier for our experiments, we define and configure two



(a) App1



(b) App2



(c) App3

Figure 3.7: Real and simulated total size of processed data streams of three graph applications

datacentres as listed in Table 3.3. Thus, two sets of these parameters are defined in simulation.properties file, where the first set is for the first datacentre and the second set is for the second datacentre.

3.7.4 Evaluation Experiments

As we mentioned earlier, two experiments are considered for our evaluation of IoTSim-Stream, which are as follows:

- Experiment 1 (Simulator Validation): Validate the correctness of IoTSim-Stream in modelling, scheduling and executing stream graph applications in Multicloud environment. This experiment presents a comparison between the amount of data streams being processed in simulated and real time (theoretical) executions for different structures of stream graph applications. Theoretical execution is a manual (hand-held) process to execute stream graph application service-by-service and collect the results for total amount of data streams being processed by this application, providing a rigor-

ous results to compare with simulated results. Using these applications in their simple form (called simple stream graph applications) rather than their complex form (called complex stream graph applications) in this comparison is sufficient. That is because conducting this comparison for complex stream graph applications is not only so complicated in real time using theoretical execution, but it adds more complication without any need or benefit in such validation. Thus from this comparison, we can ensure the correctness of modelling simple stream graph applications in cloud infrastructures, inferring to the correctness of modelling even more complex stream graph applications as well, since only the complexity of application structure and its performance requirements are what varies.

- Experiment 2 (Performance and Scalability Evaluation): Study of the performance of IoTSim-Stream in term of execution time, CPU and memory usage along with the total amount of processed data streams with small to medium to extremely large stream graph applications. This experiment shows the ability of proposed simulator to model, simulate and schedule not only simple stream graph applications, but even more complex stream graph applications in Multicloud environment. That makes researchers confidently study the behaviours of different structures and configuration sizes of stream graph applications for further evaluations and improvements. For example, developing new provisioning and scheduling policies, improving execution performance, and studying QoS and SLA requirements for this type of applications.

Figure 3.6 shows the structures and parameter configurations of three stream graph applications (named App1, App2 and App3) that will be used in our experiments. For Experiment 1, we use those modelled applications in their simple form as shown in this figure. While for Experiment 2, we use them in their complex form (i.e. each one of them is replicated several times to generate the complex graph structure with hundreds and thousands of nodes (services)) to assess the performance and scalability of IoTSim-Stream. As seen from this figure, each stream graph application is composed of multiple services with one or more external sources. Each external source produces output data stream per second according to its data rate (in MB/s) that will be feed into corresponding service(s). And each service in this application needs the following configurations: data processing requirement, user performance requirement, input streams and output stream.

3.7.5 Experiment 1: Validation

To validate the behaviour of IoTSim-Stream, two tests are conducted. In the first test, we undertook the theoretical execution of the three modelled stream graph applications for 20 seconds and collect the total size of processed data streams as experimental results for this real execution. While in the second test, we undertook the simulated execution of these applications on real cloud infrastructure using IoTSim-Stream for also 20 seconds and collect the total size of processed data streams as experimental results. In these experiment tests,

we use the default value of data processing rate for minimum stream unit (i.e 1MB/s) defined in IoTSim-Stream for both real and simulated executions. Thus, any stream that is larger than minimum stream unit will be divided into portions and each portion is 1MB in size. For example, if the input rate of service is 4MB/s, the stream will be divided into four portions. As well for this experiment, we pre-defined the mapping of services of modelled applications (App1, App2 and App3) on VMs, where each service has one ServiceCloudlet mapped on one VM as listed in Table 3.7. Of course, there are many possible VM mappings of services of these applications, but we only present one VM mapping and use it in these experiment tests.

As comparing the total amount of data streams being processed by each modelled application in given time is a real indication for measuring the accuracy, we compare the collected experimental results of both real and simulated executions in order to quantify the accuracy and precision of IoTSim-Stream . Figure 3.7 shows the real and simulation results for modelled stream graph applications. Certainly, the increase in time leads to an increase in the amount of data streams being processed by a stream graph application. The difference between both results is very slight and the results of IoTSim-Stream simulation match very closely to the real ones. As the time increases, the little difference occurred between both results is reduced and the simulation results become more closer to match real ones. Consequently, the accuracy of simulation results from IoTSim-Stream in comparison with theoretical results is indicated that IoTSim-Stream is efficient in modelling and simulating the execution of different structures of stream graph applications on real Multicloud environment.

3.7.6 Experiment 2: Performance and Scalability Evaluation

As we mentioned before, the aim of this experiment is to analyse the overhead and scalability of CPU and memory usages as well as measuring the execution time of IoTSim-Stream simulations along with the total amount of data streams being processed during these simulations. Thus in this experiment, we use the modelled stream graph applications (App1, App2 and App3) with varying configuration sizes (ranging from very small to extremely large) as listed in Table 3.8. Each configuration size has different number of services and DPs.

The CPU usage information is collected using built-in Java management interface for the operating system (called "OperatingSystemMXBean") on which the Java Virtual Machine (JVM) is running. This usage is measured every second during simulation time and the average value is taken. While the memory usage information is collected using Java Runtime. The execution time is the time required to simulate given application at a given simulation time. Each test was repeated 10 times and average results are obtained and used in representation of experimental results. The provisioning and scheduling policy presented in Algorithm 1 is used to schedule each configuration size of each application on SVMs, where the scheduling plan for each one is the same across all ten repeated simulations. The default

Table 3.8: Number of Services and DPs in Each Configuration Size for Each Modelled Stream Graph Application

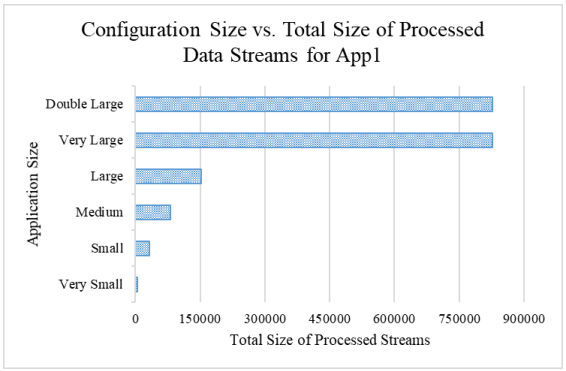
Size	**App1**	**App2**	**App3**
Very Small	4 Services - 1 DP (called App1_verysmall)	4 services - 1 DP (called App2_verysmall)	7 services - 2 DPs (called App3_verysmall)
Small	20 services - 5 DPs (called App1_small)	21 services - 5 DPs (called App2_small)	17 services - 4 DPs (called App3_small)
Medium	52 services - 13 DPs (called App1_medium)	53 services - 13 DPs (called App2_medium)	45 services - 12 DPs (called App3_medium)
Large	100 services - 25 DPs (called App1_large)	100 services - 24 DPs (called App2_large)	108 services - 30 DPs (called App3_large)
Very Large	1000 services - 250 DPs (called App1_verylarge)	1001 services - 248 DPs (called App2_verylarge)	1002 services - 282 DPs (called App3_verylarge)
Double Large	2000 services - 500 DPs (called App1_doublelarge)	2001 services - 496 DPs (called App2_doublelarge)	2001 services - 564 DPs (called App3_doublelarge)

value of data processing rate for minimum stream unit (i.e 1MB/s) defined in IoTSim-Stream is also used in this experiment.

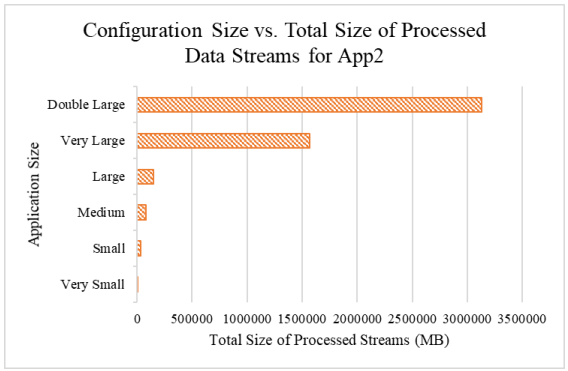
Experimental Tests under Fixed Simulation Time

The first set of tests are aimed at evaluating performance and scalability of IoTSim-Stream with different configuration sizes of the modelled applications when the simulation time is set to 5 minutes. Prior to analysis the obtained performance and scalability results, it is worth discussing the experimental results for the total amount of data streams being processed by modelled applications with their different configuration sizes. This discussion gives an indication about the amount of computations that carried-out and helps to quantify the performance of IoTSim-Stream by magnitude of processed data streams. Figure 3.8 shows the experimental results for total size of data streams being processed by each configuration size of each modelled application. From this figure, it is clear that as the configuration size of application increases the amount of processed streams is increasing, where the total size of processed streams reaches about 3TB with App2_doublelarge and App3_doublelarge for 5 minutes simulation. The exception from this increasing is App1_doublelarge since this application is linear and replicating it is also in linear way, and as simulation time is set to 5 minutes, the additional 1000 services from the prior configuration size did not process any streams (i.e. they are waiting for them). Therefore, the total amount of processed streams for this configuration size is the same as App1_verylarge.

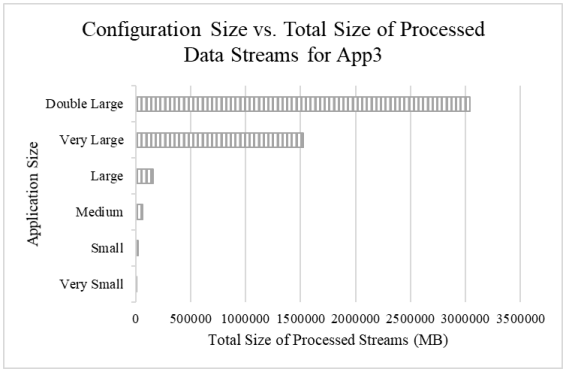
Another point from Figure 3.8 is that the total amount of streams processed by App3 in comparison with App2 is approximately the same in some cases and less in other cases particularly from small configuration size, even though App3 has a close or even more number of services and its parameter configurations shown in Figure 3.6 indicated that the total amount of streams being processed per second by its services according to user performance requirements is also greater. The reason behind it is that by considering the number of services of



(a) App1



(b) App2



(c) App3

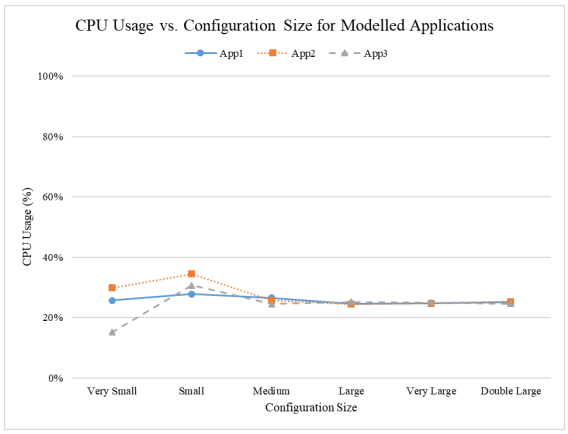
Figure 3.8: Total size of processed data streams with different configuration sizes of the modelled applications

this application (i.e. 7 services), the replication of App3 several times to reach the number of services required at each configuration size makes the total number of services being replicated is less than those services being replicated in App2, where some more intermediate and final merging services are needed to merge outputs of replicated services and produce outputs as original application. For example, to generate App3_small and App3_verylarge, App2 services are replicated 2 times and 141 times (i.e. # of services be 14 and 987) respectively, while to generate App2_small and App2_verylarge, App2 services are replicated 5 times and 248 times (i.e. # of services be 20 and 992) respectively, and the rest service(s) is/are added as intermediate and final merging services (i.e. 3 for App3_small, 15 for App3_verylarge, 1 for App2_small, and 9 for App2_Large). Overall, the amount of data being processed by those applications is huge and IoTSim-Stream is simulating them effectively.

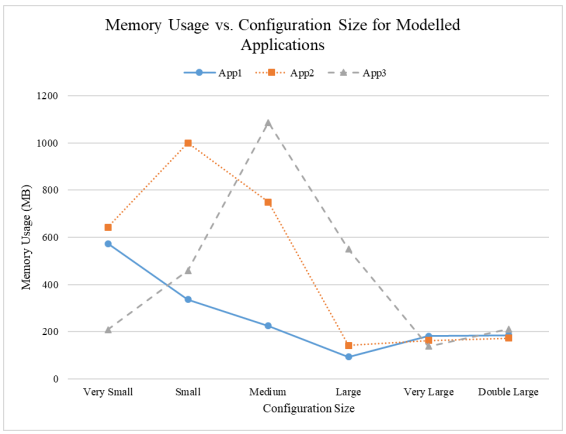
The performance and scalability results for modelled stream graph applications with their different configuration sizes are depicted in Figure 3.9. From the experimental results shown in this figure, our analysis and findings are summarised as follows:

- The results of execution time showed that the execution time is slightly increasing from very small to large configuration sizes with all modelled applications, where IoTSim-Stream is able to simulate large configuration size of App1, App2 and App3 for 5

(a) Execution Time



(b) CPU Usage



(c) Memory Usage

Figure 3.9: Performance of the modelled applications with different configuration sizes

minutes in approximately 6 seconds, 5 seconds and 5 seconds respectively and using less than 560MB of memory, where the total size of processed streams is approximately 149GB by App1, 148GB by App2 and 157GB by App3. While for very large and double large configuration sizes of the modelled applications, the execution time is significantly increased. This behaviour is expected as the number of services is 10 times and 20 times more than the number of services in large configuration size respectively as well as the total amount of streams being processed by those applications is also sharply increased. As an instance, IoTSim-Stream simulates App2_verylarge and App2_doublelarge for 5 minutes in approximately 1.8 minutes and 6 minutes respectively and using less than 200MB of memory, with total amount of processed streams is approximately 1.5TB by App2_verylarge and 3TB by App2_doublelarge. Thus, IoTSim-Stream is able to simulate a complex stream graph application with thousands of services that process huge amount of data streams (big data) with excellent performance and scalability.

- The results of CPU usage for all modelled applications with all configuration sizes except very small and small configuration sizes is not exceed 27%. This usage is an

excellent CPU performance in the machine that has 4 logical processors where this experiment is conducted, which translates to roughly usage of one logical processor. Certainly, more computing power allocated to VM leads to further decline in CPU usage. For CPU usage with very small and small configuration sizes of modelled applications, the percentage is little higher as the simulation of these applications is completed in less than 2.2 second, so that some of measured usages are CPU bursts.

- The results of memory usage showed that memory fluctuates with different configuration sizes of different modelled application. These results also showed that IoTSim-Stream is able to simulate double large configuration size of modelled applications used less than 220MB of memory. These showed that IoTSim-Stream is able to simulate complex stream graph applications with little memory overhead.

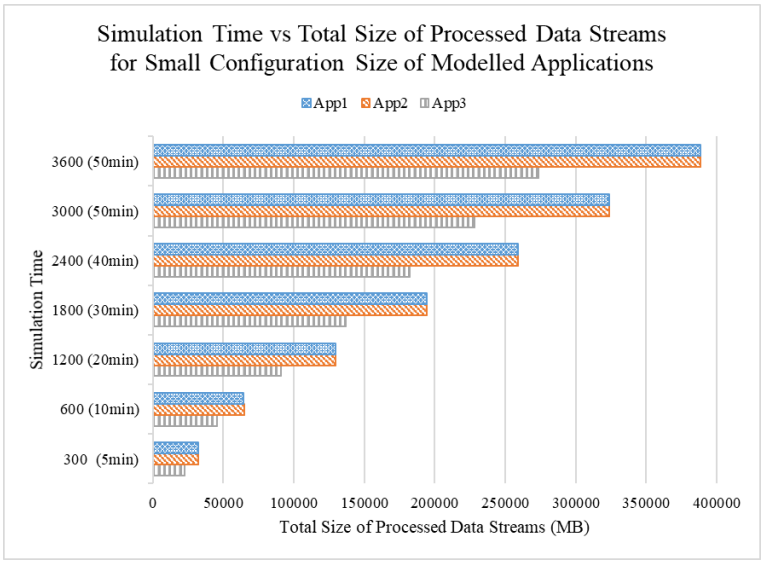
- IoTSim-Stream not only provides the ability to simulate different stream graph applications, it also offers significant gains in regards to easily measure and evaluate the execution performance. These gains are very important as it is almost unattainable to calculate and collect the execution time and performance (in term of CPU and memory usage) in a large-scale test environment on Multicloud environment.

Experimental Tests under Varying of Simulation Times

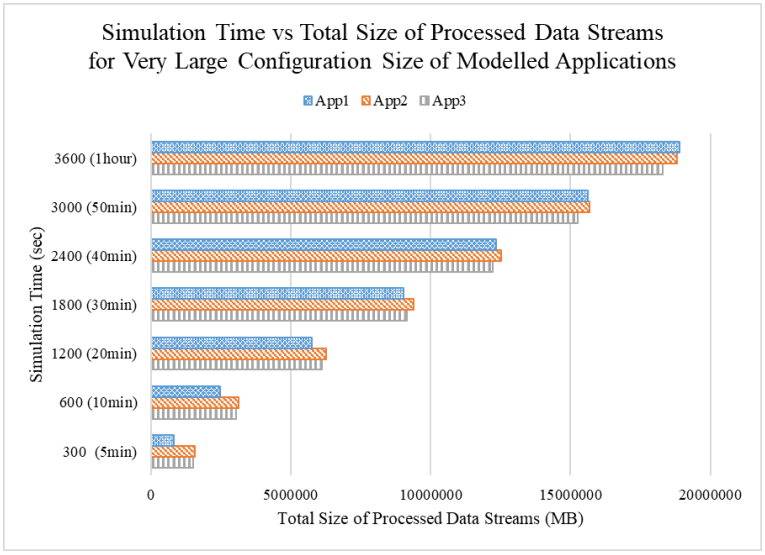
The second set of tests are aimed at evaluating performance and scalability of IoTSim-Stream with chosen configuration sizes for the modelled applications when simulation time is varying. For these tests, we chose two configuration sizes to study non-complex and complex structure of the modelled stream graph applications with the following simulation times: 300 (5min), 600 (10min), 1200 (20min), 1800 (30min), 2400 (40min), 3000 (50min) and 3600 (1hour).

Figure 3.10 depicts the total amount of streams processed by chosen configuration sizes of the modelled applications. As expected, the amount of processed data streams is increased as simulation time increases for all modelled applications, where the maximum total size of processed streams with small configuration size is approximately 379.6GB for App2, and with very large configuration size is 18TB for App1 in 1 simulation hour. That is showing how the amount of streams being processed is huge and IoTSim-Stream is effectively simulating those applications on Multicloud environment.

Figure 3.11 depicts the performance and scalability results of chosen configuration sizes of the modelled applications. The results showed that the execution time with small configuration size of all modelled application except App3 is scaled sub-linearly as simulation time increases, where IoTSim-Stream is completed 1 hour of simulation for small configuration size of App1 in less than 17 seconds, App2 in less than 15 seconds and App3 in less than 8 seconds. These performance observations for execution time with small configuration size showed that IoTSim-Stream is effectively simulating those applications for long simulation times in a very short time, within a matter of seconds. For very large configuration size of



(a) Small Configuration Size

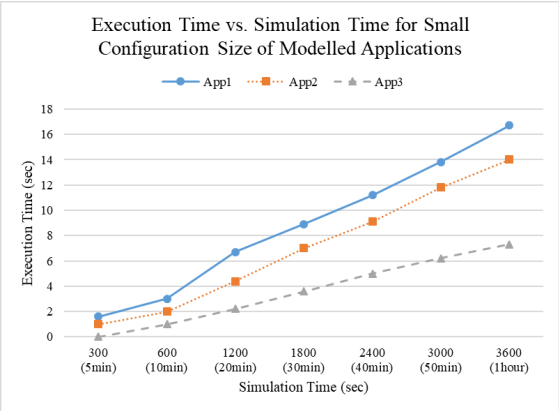


(b) Very Large Configuration Size

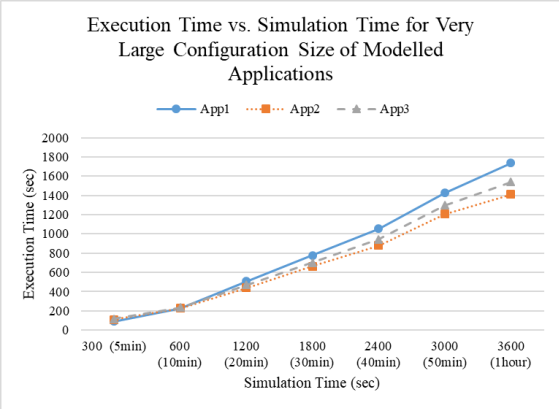
Figure 3.10: Performance evaluation with small and very large configuration sizes of the modelled applications

all modelled application, the results showed that the execution time is scaled sub-linearly as simulation time increases, where IoTSim-Stream is completed 1 hour of simulation for very large configuration size of App1 in less than 29 minutes, App2 in less than 24 minutes and App3 in less than 27 minutes. These performance observations for execution time with very large configuration size of modelled applications showed that IoTSim-Stream is effectively simulating those complex applications for long simulation times in a short and reasonable time, within a matter of minutes.

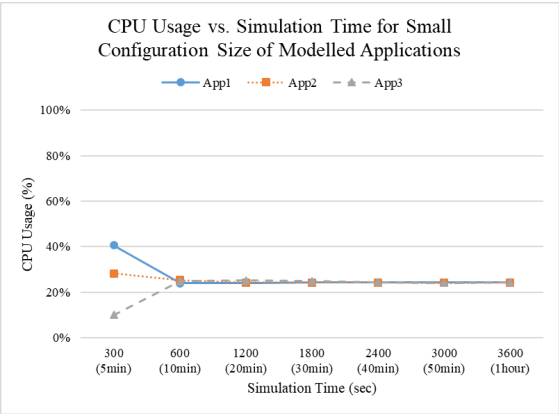
In regards to CPU usage with small configuration size, we observed that a fluctuation between approximately 10% and 41% for modelled applications when simulation time is



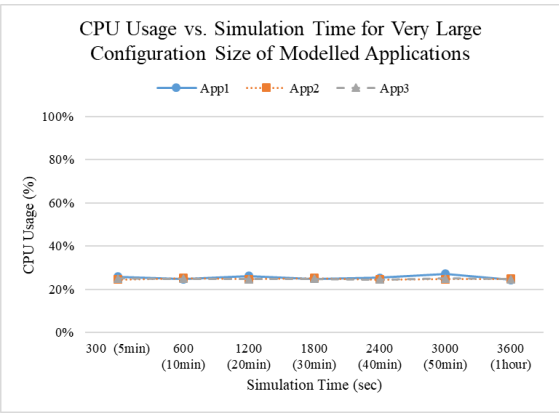
(a) Execution Time (Small Configuration Size)



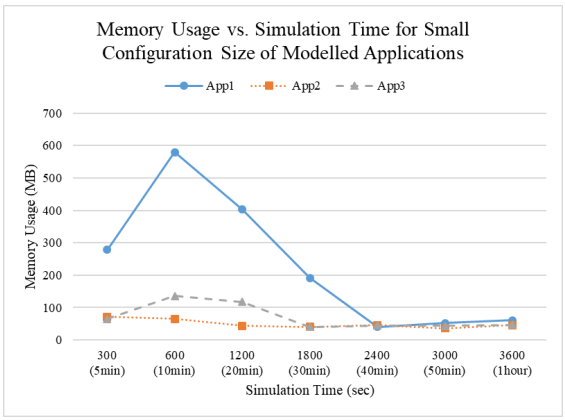
(b) Execution Time (Very Large Configuration Size)



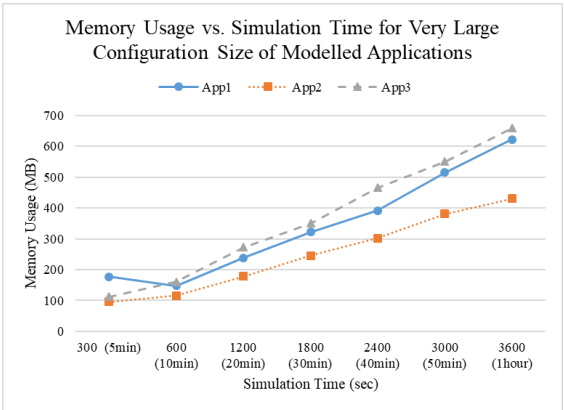
(c) CPU Usage (Small Configuration Size)



(d) CPU Usage (Very Large Configuration Size)



(e) Memory Usage (Small Configuration Size)



(f) Memory Usage (Very Large Configuration Size)

Figure 3.11: Performance of the modelled applications with small and very large configuration sizes

5 minutes. As simulation time increases, we observed a steady usage and this usage is not exceed 26%. While with very large configuration size, we observed a steady usage as simulation time increases and the this usage is not exceed 28%.

As regards memory usage with small configuration size, the results showed a significant dropping in this usage for App1 at 40 minutes of simulation and slight dropping in this usage for App2 and App3 at 30 minutes of simulation due to the behaviour of modelled application, and then it becomes steady as simulation time increases with all modelled applications. The lowest memory usage recorded with small configuration size is 36MB. While with very large configuration size, we observed that memory usage is scaled sub-linear and never grew beyond 660 MB even for 1 hour of simulation. Therefore, less than 700MB of memory is sufficient for IoTSim-Stream to simulate very large configuration size of each modelled application for 1 hour, where each application processed several terabytes of data streams during this simulation.

3.8 Significance and Practicality of IoTSim-Stream

To have a look on the practicality of the proposed simulator, we discuss one of IoT graph applications in smart cities as a real world example. Connected cars application has become largely and widely accepted. By 2020, Gartner foresees more than a quarter billion connected vehicles on the road, where each one of them produces approximately 25GB of data per driving hour [4]. Analysing the flood of data coming from roadside infrastructure (e.g. traffic lights, cameras) and connected cars allow to get real-time analytical insights that help in different services of smart city such as traffic condition and control, and smart parking. Modelling such type of IoT application using IoTSim-Stream is a straightforward task to investigate how this application will behave and evaluate its performance in cloud infrastructures at no execution cost.

In this IoT graph application, each roadside infrastructure device or connected car can be modelled as an external source, and each analytical component (such as vehicle detection, roadside data analysis, traffic analysis and traffic controlling) can be modelled as an independent service and is executed over any virtual resources. The coordination of application execution (i.e. control flow) and data dependencies (i.e. data flow) among the modelled services are defined in accordance of application logic. Based on that, the flows of data from external sources are continuously injected into the corresponding services and those flows from internal sources as continuous output streams which are results of the continuous computations carried-out by modelled services are routed towards the corresponding services.

The structure of this graph application involves heterogeneous services, multiple data sources, multiple input and output streams, can now be expressed in DAG file by including all modelled services with their data processing requirements and performance constraints that defined by the owner of this application, and data dependencies among them. Moreover, IoTSim-Stream supports the modelling of different patterns/structures of stream workflow

applications, which are linear, branching and hybrid. Linear workflow pattern (like App1) is a multi-stage application, where each stage processes input stream generated by the previous stage and produces the output stream to the following stage. Branching workflow pattern (like App2) is an application with limited precedence constraints that splits data stream to perform different parallel processing and then combing the results for further analysing. Hybrid workflow pattern (like App3) is a mix of linear and branching patterns. Thus, whether the pattern/structure of the aforementioned IoT graph application is linear, branching or hybrid with various data processing requirements and configuration complexities, IoTSim-Stream is able to simulate it in Multicloud environment. Furthermore, IoTSim-Stream enables the researchers to define the execution environment with its network performance (i.e. Multicloud environment), providing the full capability to study and investigate the performance of this application in cloud computing platforms.

Accordingly, the aforementioned real world example illustrates the need for modelling and orchestrating IoT graph application in simulation environment to support experiments at the planning phase to further enhance and improve prior to being deployed in real cloud infrastructures at the production phase. By controlling the configurations of graph application, execution environment and simulation environment, the difficulty of handing over the power of real-time data analytics is simplified even with the most complicated and distributed data pipelines. Thus, the requirements of achieving real-time data analysis and efficient workflow orchestration can be investigated through controllable and repeatable experiments, leading to further research studies including proposing resource and scheduling policies that adheres to user-defined SLA and QoS requirements, improving performance and minimising execution cost - that is what the generalised IoTSim-Stream aims to provide.

From the above discussion, our proposed simulator offers significant benefits to researchers, allowing them to (1) study how stream graph applications will perform in the cloud and its performance, (2) evaluate the efficiency of new scheduling and resource allocation policies for such applications in a real-world simulation environment, (3) test SLA-oriented management and execution optimisation of stream graph applications in cloud infrastructures free of cost, and (4) tune the performance bottlenecks at the planning and testing stage prior to production by deploying the stream graph application on multiple commercial cloud platforms. Furthermore, IoTSim-Stream is designed to be an extensible and customisable simulation toolkit, so that it provides the ability for researchers to extend and define their policies for adhering to user-defined SLA and QoS requirements, and execution optimisation. It also allows them to extend and define policies in all components of the CloudSim software stack since it was built on top of CloudSim. IoTSim-Stream is a valuable research simulation toolkit that deals with both complexities emerging from modelling stream graph application and simulated environments. With IoTSim-Stream, both research and industry communities can reduce the time needed to evaluate the new designs and scenarios, where these scenarios can be evaluated in hours or days instead of weeks and months.

3.9 Summary

In this chapter, we proposed IoTSim-Stream, a simulation toolkit for modelling and executing stream graph applications in a Multicloud environment. We also presented the main components of IoTSim-Stream with their functionalities. IoTSim-Stream provides fully custom simulation parameters, making it a suitable research tool to assist researchers in simulating and studying the behaviour of stream graph application in cloud computing environments with easy to set-up Multicloud environment and customisable user performance requirements. The efficiency of the proposed IoTSim-Stream in simulating various structures of stream graph applications has been evaluated by conducting a comparison between theoretical and simulated executions. Moreover from the results of extensive performance and scalability evaluations, we found that IoTSim-Stream is efficient for simulating different patterns/structures of stream graph applications with various data processing requirements and configuration complexities.

Following the development and implementation of IoTSim-Stream, we need to investigate the scheduling problem of stream workflows in a Multicloud environment. Thus, the next chapter will discuss how to schedule the stream workflow efficiently over cloud infrastructures at deployment time while meeting user real-time data processing requirements and minimising total execution cost.

Chapter 4

Meta-Scheduling for Efficient Stream Workflows Execution

The execution of stream workflow applications on cloud environments requires advanced scheduling techniques that adhere to end user's requirements in terms of data processing and deadline for decision making. In this chapter, we propose two scheduling and resource allocation techniques for efficient execution of stream workflow applications in Multicloud environment while adhering to workflow application and user performance requirements and reducing execution cost. These algorithms address offline scheduling by making scheduling decisions before the execution of stream workflow applications, assuming no change can be made to these applications. Results showed that the proposed genetic algorithm is effective for all experiment scenarios.

4.1 Introduction

Given the complexity and heterogeneity of stream workflows and the compute resources in addition to user-defined QoS requirements, stream workflow scheduling is a new class of scheduling problem. The execution of the stream workflow application over cloud infrastructures is not a trivial task. However, most research works have focused on big data batch computing and thus big data stream computing is still receiving little attention. Thus, few scheduling methods found in the literature that are related to our work.

Looking at the streaming operator graphs used with stream processing systems, it is important to determine the differences between these graphs and stream workflows that we consider in this chapter, and therefore how our scheduling problem is different. Stream-oriented big data platforms and services such as Apache Storm and Azure Stream Analytics provide the ability to design streaming operator graphs to process streams and produce a final output stream. First of all, streaming operator graphs generated by those systems differ from stream workflows as there is one source of data for the whole operator graph and there is one end operator, while a stream workflow has multiple input data sources and multiple output streams. Moreover, each component in a stream workflow has heterogeneous platform and infrastructure requirements. Furthermore, the goal of these systems is to attain low stream latency without taking into consideration other optimisation goals such as network usage, execution performance and cost.

In regards to the most related research works (Pietzuch et al., 2006), (Cardellini et al., 2016) and (Venkataraman et al., 2017), these works also addressed the placement problem for those operator graphs in distributed large-scale environments with various limitations. (Pietzuch et al., 2006) proposed an algorithm (called SBON) that optimises operator placement to enhance network utilisation by using the continuous knowledge of network and node conditions (i.e. network usage metric), aiming at providing low latency. This research work lacks the consideration of the location of data stream sources in making placement decisions. In the same context, (Cardellini et al., 2016) proposed an optimal placement model and prototype scheduler for operator graphs that optimised user-oriented QoS attributes. This work only presented the modelling of network-related QoS attributes (elastic energy, network usage and inter-node traffic), and made the consideration of other constraints such as execution cost or performance as future research directions. It also ignores the user-defined performance constraints on the operator graph. (Venkataraman et al., 2017) focused on optimising the scheduling of operator graph and presented techniques that are implemented in Drizzle to enable high throughput and adaptability, and low latency. This work ignores the consideration of data source location, relies on micro-batch processing system (i.e. Apache Spark) to provide stream processing at scale. It also lacks the consideration of user-oriented QoS attributes. Accordingly, the placement problem of operator graph is related to a different type of stream graph application as well as having different assumption and optimisation goals in comparison to the stream workflow and scheduling problem that we consider in this

chapter. The stream workflow scheduling problem considers the mapping of each analytical component to one or more compute resources as well as the optimisation goals are minimising execution cost and improving performance without violating real-time user requirements.

In accordance to the above overall discussion, scheduling and resource allocation technique is needed for stream workflow applications. To this end, we design and implement two efficient scheduling algorithms using Greedy and Genetic heuristics. We evaluate their efficiency by comparing them using commonly types of real workflow structures in different experiment scenarios and present experimental results.

This chapter is structured as follows: Section 4.2 presents Multicloud environment, and stream workflow and its requirements. Section 4.3 presents our problem modelling and the terminology used, while in Section 4.4, we explain in detail the proposed resource provisioning and scheduling algorithms. Section 4.5 presents our experiment methodology to evaluate the efficiency of the proposed algorithms and discusses the obtained results. Section 4.6 concludes the chapter.

4.2 Multicloud Execution Environment and Stream Workflow Application

4.2.1 Overview of Multicloud Environments

When targeting distributed data sources that inject their data streams into a workflow pipeline, it is necessary to utilise data locality by leveraging a Multicloud architecture. If all resources are provisioned from a single cloud and not all data sources are near this cloud, the transfer of large amounts of data to corresponding resources not only leads to the difficulty of achieving the requirements of real-time data analysis, but is also expensive and incurs high latency. Moreover, if the location of the data source changes at any time, the flexibility provided by a Multicloud architecture allows the corresponding analytical component to be moved to a new data location. Furthermore, if the amount of data produced by a data source decreases overtime and reaches low data rate, the opportunity to move the corresponding analytical component to another cloud helps to improve performance and reduce the cost without violating user-defined real-time requirements. A single cloud cannot deal with all of the aforementioned points, and thus a Multicloud architecture should be preferred in these scenarios.

A global view of a Multicloud environment is depicted in Figure 4.1. Each cloud is independent from other clouds and offers different levels of compute capacity at different costs. The network bandwidth between compute resources in one cloud is mostly unchanged, but is variable between various clouds. Similarly, the latency between compute resources in one cloud is mostly low, while between various clouds, it can be comparatively high.

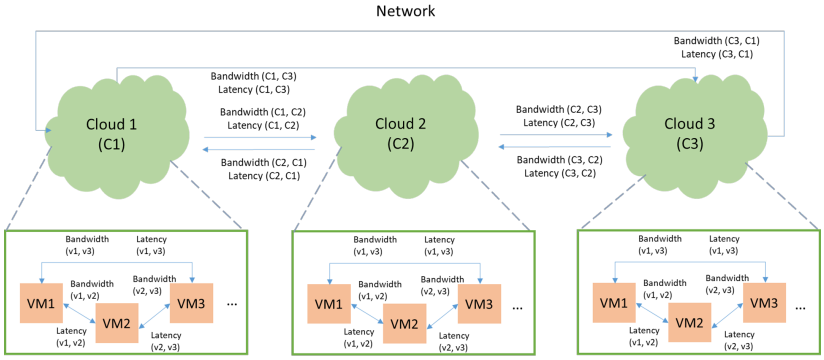


Figure 4.1: Multicloud environment: Network.

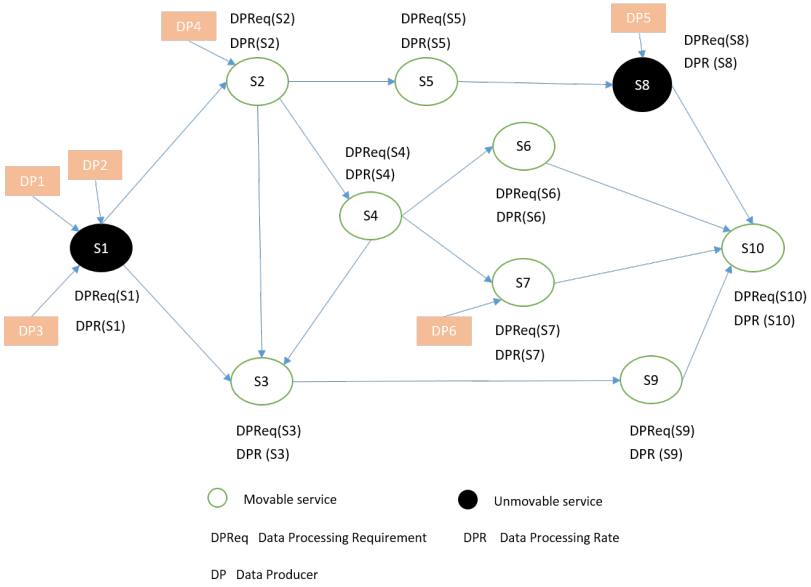


Figure 4.2: Stream workflow application example.

4.2.2 Stream Workflow Applications and their Requirements

Stream workflow applications comprise multiple streaming analytical components, which can be seen as services, as they can independently execute over any virtual resources, although data dependencies among them should be maintained. With this workflow application, we deal with continuous inputs from internal sources (i.e. output data of parent services) as well as from external sources (such as sensors), continuous data processing that is carried out by running services for incoming data and continuous outputs that are results of processing data at services, which routed towards one or more child services. The end services generate the continuous output results for the execution of this workflow. Figure 4.2 shows an example of stream workflow application with its requirements. With this workflow application, the two types of services are:

- Unmovable service: It is a service with unmovable data, which means the data volume coming from data stream sources is large and we need to process such data locally to avoid the cost and time of transfer data. Thus, data locality approach is applied with

this service; or

- Moveable service: It is a service with movable data, which means the working stream is small and can be transferred with low communication overhead of data transmission. Thus, placement optimisation approach is applied to exploit deployment flexibility.

As noted in Figure 4.2, each service has its own data processing requirements, which is the number of instructions required to process one MB of stream data, and data processing rate, which is the measure of the amount of data that can be processed in a given time by a service (in MB/s). In term of the mode of data that being routed towards one or more child services, there are two data modes:

- Replica mode: The child service receives replica copy of the output stream of a parent service.

- Partition mode: The child service receives a portion of the output stream of a parent service according to the specified partition percentage.

The owner of stream workflow application allows to specify maximum performance constraints in terms of data processing rate of services targeting the maximum desired processing performance that she/he is willing to pay for achieving it during the whole execution, and letting the cost minimisation carried-out at initial scheduling plan. If no performance constraints are specified, the initial input data rates of services are considered as the maximum performance constraints (representing maximum desired processing performance for those services). Of course, the input data rate is varying overtime, so that a strategy is needed to handle the increase in data rate. We assume that the exceed incoming data rate will be dropped, thus the increase of load above the pre-specified maximum throughputs will have no effect. Of course, if the speed of incoming data streams decreases, the throughput of service still has the full capability to handle the increase in the speed of data upto the pre specified processing performance. In addition to achieving user specific performance constraints in term of throughputs of services, the end-to-end latency (response time) is crucial in stream workflow application. It is the time between receiving a data stream at a service and generating output stream that regards this stream. Ensuring the low latency is required during the whole execution of stream workflow. It should be kept as low as possible or be bounded when it starts to increase whilst maintaining user specific throughput.

Accordingly, the variables of stream workflow are service type, its data processing requirement, its data processing rate and the dynamism of execution environment. The latter includes network bandwidth and latency between different clouds. As a result, both user performance requirements and workflow application requirements need to be considered and achieved in addition to maintaining low latency during the execution of this workflow.

Table 4.1: Problem Modelling Notation

Symbol / Term	Description
G	Workflow graph
S	Set of all graph services
E	Set of all graph edges
¥^{m}	Percentage of data that is routed from parent service to child service (100% in replica mode or any percent in partition mode)
S_n	Particular service in workflow graph
MI^{S_n}	Number of floating-point operations required to process one MB of input data (MI/MB)
λ^{S_n}	Amount of data produced by a given external sources(s) and being consumed by a service (MB/s)
γ^{S_n}	Proportion of output data to input data for S_n
C	Set of all clouds in Multicloud environment
c_g	Particular cloud in Multicloud environment
L	Network latency matrix
B	Network bandwidth matrix
DTCOST	Data transfer cost matrix
VM^g	Set of all VMs in cloud g
vm^g_k	Particular VM k in cloud g
U^g	Set of all internal network links between VMs in cloud g
u^g_h	Particular internal link between vm^g_{org} and vm^g_{dest}
$MIPS_{vm^g_k}$	Rating of the capacity of VM k in cloud g
ċ$_{vm^g_k}$	Provisioning cost of VM k in cloud g (cents/s)
μ^{S_n}	User-defined maximum performance constraint (MB/s)
α_{S_n}	Data processing rate of S_n
unitDUnit	Minimum stream unit for the whole application (MB)
unitDPRate	Minimum stream unit per second for the whole application (MB/s)

4.3 Problem Modelling

Prior to introduce the problem modelling of stream workflow application, we list all the terminologies that will be used in this model in Table 4.1.

4.3.1 Application Model

We model a stream workflow application as a DAG $G = (S, E)$. S represents a set of N services $S = s_1, s_2, ..., s_N$ and E represents a set of M edges/links between services denoted as $E = e_1, e_2, ..., e_M$. Each edge, e_m is represented as a tuple $(s^m_{org}, s^m_{dest}, ¥^m)$, where s^m_{org} denotes origin service, s^m_{dest} denotes destination service and ¥^{m} denotes the percentage of data generated by s^m_{org} that is routed towards s^m_{dest}.

Each particular service S_n, is represented as a tuple $S_n = (MI^{S_n}, \lambda^{S_n}, \gamma^{S_n})$, where MI^{S_n}

denotes the number of floating-point operations required to process one MB of incoming data (service data processing requirement) in MI/MB, λ^{S_n} denotes the arrival rate of streams generated by data sources outside the application in MB/s (such as data streams generated by sensors) to be consumed by the service, and γ^{S_n} denotes the proportion of data generated by a service based on input streams.

Notice that, given the nature of stream workflow applications, it is possible that data generated by one service can be sent to one or more services, or can be split among different services. Thus, for service S_n, both parameters γ^{S_n} and ¥^{m} (in edges where such service is origin service) are necessary to define the whole application. In addition, to process streams that coming at different speeds, the minimum stream unit per second (denoted as unitDPRate) is needed to be specified for the whole application, so that each provisioned compute resource must process at least one unit per second and of course it can process multiple units per second according to its computing capacity per second (in term of MIPS). By specifying minimum stream unit per second for the whole workflow application, the data processing rate (MB/s) for processing this unit can be determined to ensure that each provisioned compute resource at least processes one unit per second.

4.3.2 System Model

The cloud system is modelled as a tuple W = (C, L, B, D). A set of G clouds in the Multicloud environment is denoted as $C = c_1, c_2, ..., c_G$. L, B, and D denote matrices containing respectively the latency (in seconds), the bandwidth (in MB/s), and the data transfer cost (in cents/MB or ¢/MB) between each of the pair of clouds in C.

Each cloud, c_g is represented as a tuple (VM^g, U^g), where $VM^g = vm_1^g, vm_2^g, ..., vm_K^g$ is a set of K virtual machines (compute resources) with different resource configurations deployed in c_g, and $U^g = u_1^g, u_2^g, ..., u_H^g, u_h^g = (vm_{org}^g, vm_{dest}^g)$, a set of H links that are part of the datacenter network topology.

Each VM deployed in the cloud, vm_k^g, is represented as a tuple $(MIPS_{vm_k^g}, ¢_{vm_k^g})$, where $MIPS_{vm_k^g}$ denotes floating-point operations computed by this VM according to its compute capacity per second and $¢_{vm_k^g}$ denotes the cost of provisioning such VM (in cents per second).

The data processing rate for S_n if it is mapped to vm_k^g is denoted as φ_k^g and is calculated as:

$$\varphi(S_n, vm_k^g) = \frac{\lfloor MIPS_{vm_k^g}/\chi \rfloor * \chi}{MI^{S_n}} \text{ MB/s} \tag{4.1}$$

$$Where\ \chi = unitDPRate * MI^{S_n} and\ MIPS_{vm_k^g} \geq \chi$$

The workflow application owner can specify maximum performance constraint for service S_n (denoted as μ^{S_n}) as a part of request (in MB/s) as a value for data processing rate of service S_n (denoted as α_{S_n}), targeting the maximum desired processing performance that she/he is willing to pay for achieving it during the whole execution. If no performance con-

straint for service S_n is specified, the system will calculate this rate based on input stream(s) of service S_n. In that case, each service S_n is capable to handle upto the specified data processing rate (throughput) and when the speed of input streams increases this maximum throughput μ^{S_n}, the dropping mechanism is applied. Of course, if the speed of incoming data streams decreases, S_n still has the full capability to handle the increase in the speed of data upto μ^{S_n}. Let $pro(S_n)$ be a set of VMs that are provisioned from one cloud for service S_n and $inStream(S_n)$ denote the input stream of S_n.

The $inStream(S_n)$ is calculated as follows:

$$\begin{aligned} inStream(S_n) &= \lambda^{S_n} + \\ &\sum\nolimits_{e_m \in E | s^m_{dest} = S_n} (\gamma^{s^m_{org}} * \sum\nolimits_{v \in pro(s^m_{org})} \varphi(s^m_{org}, v)) * ¥^m \text{ MB/s} \end{aligned} \tag{4.2}$$

The following constraint of data processing should be maintained:

$$\sum\nolimits_{v \in pro(S_n)} \varphi(S_n, v) \geq \alpha_{S_n} \tag{4.3}$$

$$Where\ \alpha_{S_n} = \begin{cases} \mu^{S_n}, if\ maximum\ throughput \\ inStream(S_n), otherwise \end{cases}$$

Additionally, we assume that every data stream should be processed, as unprocessed data streams lead to incorrect results. We also assume that the order of stream portions should be maintained during the distribution among corresponding compute resources. Based on these assumptions, we maintain user specific throughputs for all services, and end-to-end latency (response time) as low as possible or even bounded when it is being increased, because if the input data rate of a service exceeded the data rate specified in processing performance constraint, the exceeded streams will be dropped. Thus, the incoming data streams upto throughput of a service are processed as they arrive, and the latency from the time of stream being added to input queue until its emission from the service as output stream is maintained. Of course, in case of a child service receives two or more dependency streams from its parents services, the latency is from the time of the last dependency stream being added to input queue until its emission from child service.

Each service S_n in workflow application produces output stream as a result of computation. Let $outStream(S_n)$ denotes the output data stream for a service S_n and is calculated as follows:

$$outStream(S_n) = \gamma^{S_n} * inStream(S_n) \qquad \text{MB/s} \tag{4.4}$$

The total cost of running all provisioned VMs for all services to process incoming data streams during the period of time T (which represents a set of I one-second intervals, $T = t_1, t_2, ..., t_I$), is denoted as execCost(S,T) and is calculated as:

$$execCost(S, T) = T * \sum\nolimits_{S_n} \sum\nolimits_{v \in pro(S_n)} ¢_v \qquad \text{cents} \tag{4.5}$$

The data transfer cost is based on the amount of data being moved, the cost of data transfer charged by cloud provider, and network performance. In a workflow application, both input and output data are moved among different clouds. As the speed of data may vary during workflow execution either decreases below service throughput or increases upto service throughput (as exceed load will be dropped), the calculation of data transfer cost needs to be carried-out per second. Let $cts(S_n)$ denotes the cost of transferring streams for S_n (including input streams from other services) per second, and $CTStream(S,T)$ denotes the total data transfer cost for the amount of data being moved for all services during the period of time T. The $CTStream(S,T)$ is calculated as follows:

$$CTStream(S,T) = \sum\nolimits_{t_i} \sum\nolimits_{S_n} cts(S_n) \qquad \text{cents} \tag{4.6}$$

$$cts(S_n) = \sum\nolimits_{S_i \in parent(S_n)} c(S_i) \text{ cents}$$

$$c(S_i) = \begin{cases} 0, & \text{if } C_g(S_i) = C_g(S_n) \\ outStream'(S_i) & \\ *D(C_g(S_i), & \\ C_g(S_n)), & \text{otherwise} \end{cases}$$

$$outStream'(S_i) \begin{cases} outStream(S_i), & \text{if } \varrho \leq 1 \\ \frac{outStream(S_i) * ¥^x}{\varrho}, & \text{otherwise} \end{cases}$$

$$Where\ \varrho = \frac{outStream(S_i) * ¥^x}{B(C_g(S_i), C_g(S_n))} + L(C_g(S_i), C_g(S_n))$$

, and $parent(S_n)$is the set of parent services for service S_n

Thus, the objective function is to minimise the cost of execution of workflow without compromising user performance requirements in term of maximum throughputs:

$$minf(S,T) = execCost(S,T) + ctStream(S,T) \tag{4.7}$$

4.4 Proposed Algorithms

The problem of selecting the right resources for executing stream workflow applications in Multicloud environments to meet user requirements and to achieve efficient performance (in term of throughput and latency) while minimising the costs of resource provisioning and data transfer is an optimisation problem, where resource selection problem is generally NP-complete problem. Our research problem is to find near-optimal resource selection solution with minimal execution cost at deployment time for executing stream workflow application in

Algorithm 3 Greedy Scheduling

procedure GREEDYSELECTION(VMOffers, unitDPRate)
for `each service` S_n `from S` **do**
  $selectedVMList \leftarrow \phi$
  $cost \leftarrow \infty$
  $unitMIPS \leftarrow unitDPRate * MI^{S_n}$
  $reqMIPS \leftarrow$ get MIPS based on α_{S_n} and $unitMIPS$
  for `each cloud` c_g `from C` **do**
    if S_n *is unmovable &* $c_g \neq$ `placement cloud of` S_n **then**
      continue
    end if
    $selectedVM \leftarrow 0$
    $reqUnits = reqMIPS/unitMIPS$
    $workingVMList \leftarrow \phi$
    $VM^g \leftarrow$ list of VM offers for c_g
    $VM^g \leftarrow VM^g - \{x \in VM^g \mid MIPS_x < unitMIPS\}$
    if VM^g is empty **then**
      if $S_n$is unmovable $||S_n$ is movable & c_g is last cloud **then**
        exit
      else
        continue
      end if
    end if
    while $reqUnits > 0$ **do**
      $maxVMValue \leftarrow 0$
      for `each` vm_k^g `from` VM^g **do**
        $achievedPortionsByVM \leftarrow \lfloor MIPS_{vm_k^g}/(unitMIPS) \rfloor$
        $vmValue \leftarrow (achievedPortionsByVM/reqUnits)/¢_{vm_k^g}$
        $vmValue \leftarrow vmValue + \lfloor MIPS_{vm_k^g}/(unitMIPS * \#OfServiceDependencies) \rfloor /¢_{vm_k^g}$
        if $vmValue > maxVMValue$ **then**
          $maxVMValue \leftarrow vmValue$
          $selectedVM \leftarrow k$
        end if
      end for
      $workingVMList \leftarrow workingVMList \cup \{VM^g_{selectedVM}\}$
      $acheivedPortions \leftarrow \lfloor MIPS_{vm^g_{selectedVM}}/unitMIPS \rfloor$
      $reqUnits \leftarrow reqUnits - acheivedPortions$
    end while
    $newCost \leftarrow \sum_{v \in workingVMList} ¢v$
    if $newCost < cost$ **then**
      $cost \leftarrow newCostv$
      $selectedVMList \leftarrow workingVMList$
    end if
    if S_n *is unmovable &* $c_g =$ *placement cloud of* S_n **then**
      break
    end if
  end for
  add selectedVMList of S_n to ServiceVMsMap
end for
end procedure

Multicloud environment, where the required resources are provisioned based on user-defined performance requirements and then services are being scheduled on these resources before the execution begins. For that, we propose two resource provisioning and scheduling algorithms using Greedy and Genetic heuristics.

4.4.1 Greedy Scheduling Algorithm

A greedy algorithm is a heuristic algorithm that finds the best solution at each stage (local optimum) without consideration of future results, hoping to find global optimum. For our resource provisioning and scheduling problem for executing stream workflow application in Multicloud environment, we propose a greedy algorithm that finds the best resource selection solution for a given workflow application at deployment time. The pseudocode of this proposed algorithm is shown in Algorithm 3. This algorithm takes $O(SCUV)$ with S the number of services, C the number of clouds, U the maximum number of required minimum data processing units of any service and V the number of VM offers in the placement cloud.

4.4.2 Genetic Scheduling Algorithm

For the research problem discussed in this chapter, search spaces are large and complex, with many cloud offerings available and several problem-dependent constraints to be satisfied. The search space will rapidly grow when looking for efficient schedules of increasing problem size. To deal with scheduling problem of stream workflow at deployment time, the goal is to find near-optimal solution by rapidly traversing large search spaces and generate scheduling plan for starting the execution of this workflow.

Genetic Algorithm (GA) is a useful algorithm to this problem because of its effectiveness at searching large and complex spaces to enable the practical implementation of optimising scheduling. It is capable to provide several satisfying candidate solutions (i.e. resource selection solutions) to choice from by evolving over generations of candidate solutions. Algorithm 4 shows the pseudocode of the proposed genetic resource provisioning and scheduling algorithm. This algorithm takes $O(GPS^2D)$ with G the number of generations (as termination condition), P the size of population, S the length of candidate solution (number of services) and D the maximum number of stream dependencies of any service. Our proposed GA is implemented using the Watchmaker framework for evolutionary computation (Dyer, 2010).

Encoding

For stream workflow application, each candidate (individual) in the population represents a feasible workflow resource selection solution, which is composed of a set of chromosomes. Each chromosome is a data structure in which a resource selection for a service is encoded. It contains the identifier of service (serviceID) that represented by integer number and the genes of service that represented by the list of integer numbers, i.e. identifiers of selected virtual machines from particular cloud, as depicted in 4.3b. We assume that the selected VMs for a service in candidate solution should be from one cloud, where different instances of VMs can be selected as well as the same VM can be reselected many times (reputation is allowed). To deal with multiple clouds and their VM offerings, the identifiers of VMs offered by all clouds should be globally unique, thus they can be used conveniently in genes

Table 4.2: The global VM mapping for clouds

vmgid	vmidAtCloud	CloudID	TotalMIPS
0	0	0	7000
1	1	0	13000
2	2	0	26000
3	3	0	54000
4	0	1	5500
5	1	1	11000
6	2	1	22000
7	3	1	44000
8	0	2	5000
9	1	2	10000
10	2	2	20000
11	3	2	40000

of chromosome's without any possible conflict. Therefore, we create a global VM mapping that map each VM offered by each cloud to global VM identifier. For instance, the global VM mapping listed in Table 4.2 is for the following three different clouds: Cloud 0 (contains VM0, VM1, VM2, VM3), Cloud 1 (contains VM0, VM1, Vm2, VM3) and Cloud 2 contains (VM0, VM1, VM2, VM3).

Based on the above global VM mapping for three clouds, the representation of possible candidate solution for sample workflow application is shown in Figure 4.3c. The presented encoding is equivalent to a two-dimensional integer, where one dimension represents the identifiers of services while the other shows the list of global identifiers of VMs selected that will be provisioned from particular cloud for a service. Using this encoding makes the genetic manipulations easier, for example to apply crossover, we simply swap chromosomes (services) between two candidates according to crossover points without any iteration over chromosomes genes.

Initial Population

The initial population is contained greedy solution as a one chromosome and N-1 chromosomes that are randomly generated, making the search space covering a wide range of possible resource selection solutions. For creating random chromosome, the following steps are followed in order to generate possible resource selection for a service S_n:

- Select a random cloud c_g if S_n is movable, otherwise use the placement cloud
- Compute the required MIPS for minimum stream unit per second specified in the workflow application.
- Compute the required MIPS based on owner-defined data processing rate α_{S_n} for S_n.
- Select a random VM vm_k^g from c_g.

- Add the global identifier of vm_k^g to the list of genes for S_n if $MIPS_{vm_k^g} \geq$ required MIPS for minimum stream unit per second.
- Repeat until the constraint in equation 4.3 is maintained.

Fitness Function

The chromosomes from first generation and from subsequent generations are evaluated by a fitness function that measures the quality of the solution in solving the problem at hand. As our goal is to minimise the cost of execution of a stream workflow application including resource provisioning cost and data transfer cost while maintaining the accuracy of application and achieving user performance requirements, the fitness value for a solution is computed using Equation 4.7.

Selection

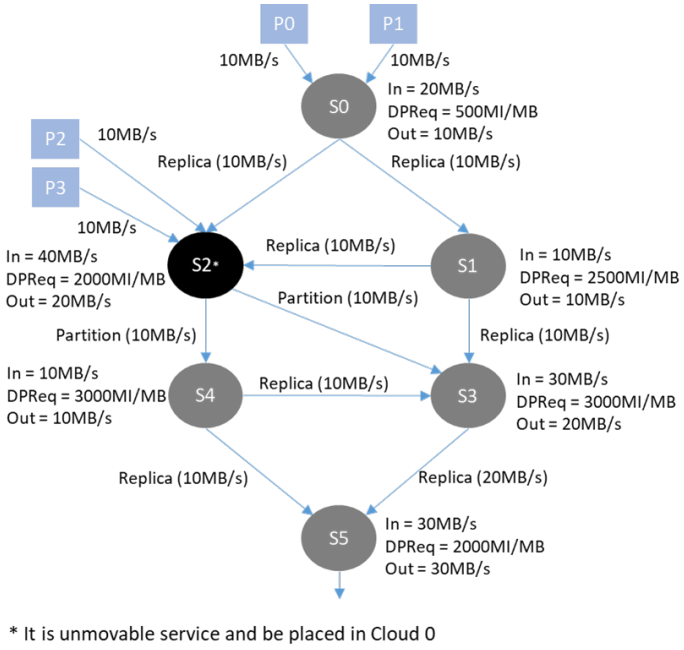
The selection operator is used to select candidate solutions that will be reproduced, creating the foundation of the next generation. This selection of some candidate solutions is according to their fitness, where the fittest solutions have better chance to be selected compared to weaker ones. However, before making any selection in each generation, elitist selection is performed, where the fittest candidate solution(s) in such generation is copied, unchanged, to the next generation.

Crossover

The reasoning of using the crossover operator is to create a new candidate solutions (named offsprings) by combing two fittest candidate solutions, which may result in even better solution for solving the problem. To maintain our assumption regarding to resource provisioning for a service (resources are provisioned from one cloud for this service in one candidate solution, possibly different clouds in different candidate solutions) and to avoid producing invalid candidate solutions, we exchange chromosomes of services between two candidate solutions. Thus, our crossover is two-point crossover operator (equivalent to twice one-point crossovers) that swaps service's chromosomes between two points among two parents, producing two new offsprings. As shown in Figure 4.4, the two candidate solutions are randomly chosen from the current population as parents, then two points are randomly chosen as crossover points (one point at index 1 and another point at index 3), and afterwards that service's chromosomes of parents are exchanged between these two points. The crossover operator is performed on the selected candidate solutions in the generation with a certain probability.

Mutation

The rationale of using mutation operator is to retain the diversity of population from one generation to the next generation. In other words, it leads the search space to escape from



(a) Sample stream workflow application

ServiceID	vmgid	vmgid	...
ServiceID	vmgid	vmgid	...
ServiceID	vmgid	vmgid	...
ServiceID	vmgid	vmgid	...
ServiceID	vmgid	vmgid	...
ServiceID	vmgid	vmgid	...

(b) General candidate encoding

S0	0	0	
S1	5	5	4
S2	3	3	
S3	3	2	2
S4	10	9	
S5	11	11	

(c) Sample candidate encoding

Figure 4.3: Candidate solution encoding for sample stream workflow application.

local optima to arrive at the global optimum. It also helps to change genetic material of certain parents.

In our problem, applying blind random changes for genes of chromosomes in a candidate solution may generate invalid solutions that violates application accuracy and performance constraints. Hence, our mutation operator is mutating the candidate solution intelligently by replacing the random gene (VM) in one of its chromosomes with new gene (new VM) that does not violate minimum data processing requirement and has lower resource provisioning cost, where different chromosomes within this candidate can be mutated with certain probability (mutation probability). In case of no such new gene is found that meets the requirement, the chromosome of candidate solution is left without mutation, and the other chromosomes of this candidate solution are subjected to mutation based on mutation probability. By doing that, no invalid candidate solution will be produced after applying mutation on the chromosomes of selected candidate solution. An example of candidate solution mutation is shown in Figure 4.5. In this example, the second gene of third chromosome and the first gene of last

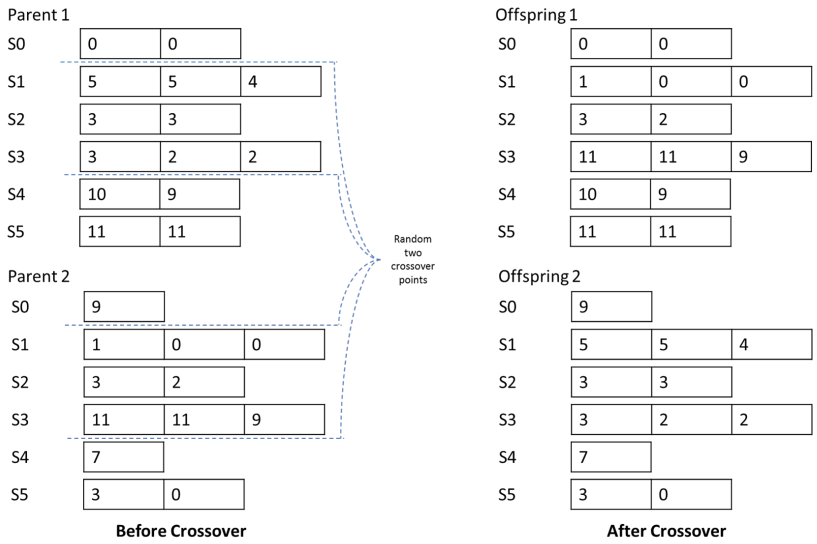


Figure 4.4: Crossover in the context of our solution.

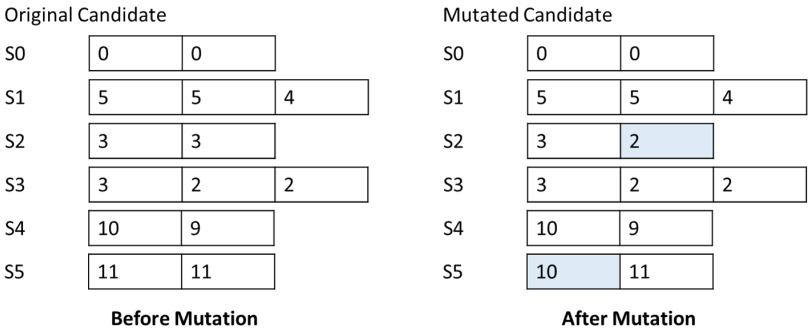


Figure 4.5: Mutation as performed by our solution.

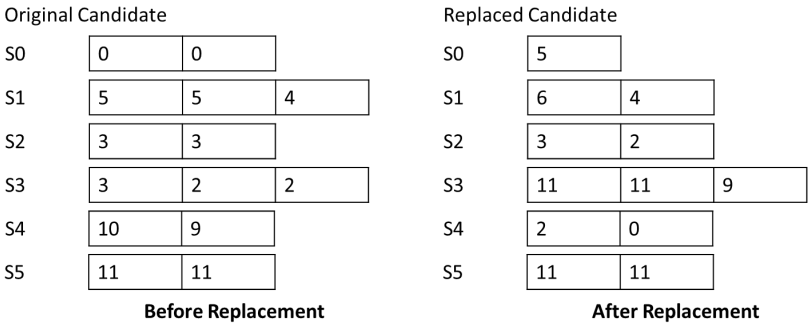


Figure 4.6: Replacement in our solution.

chromosome are mutated in this selected candidate solution, producing a mutated candidate solution.

Replacement

As with the mutation operator, the rationale of using replacement operator is to maintain genetic diversity within the population. Our replacement operator replaces those solutions from the selected candidate solutions whose fitnesses are greater than average fitness based

Algorithm 4 Genetic Resource Scheduling

```
1: P ← empty initial population
2: call greedy algorithm and add its solution to P
3: generate N-1 candidates randomly and add them to P
4: calculate fitness values for candidates in P
5: sort candidates in P in ascending order of fitness
6: while condition not satisfied do
7:    perform elitist selection
8:    select candidates using selection operator for evolving
9:    create new offsprings using crossover operator
10:   create new offsprings using mutation operator
11:   replace weakest candidates using replacement opoerator
12:   add elite candidates to the evovled population
13:   calculate fitness values for candidates of the evovled population
14:   sort candidates of the evolved population in the ascending order of fitness
15: end while
16: return best candidate (candidate with minimum cost)
```

on replacement probability. Each of those solutions is replaced with randomly generated solution if the fitness of the new solution is less than its fitness; otherwise, this solution is retained in the population. This operator tries twice to find a better solution for each candidate solution that is subject for replacement to replace it in order to keep the number of trials at an acceptable level. The replacement operator is performed on the chosen candidate solutions in the generation with a certain probability. An example of candidate solution replacement is shown in Figure 4.6.

4.5 Performance Evaluation

4.5.1 Experiment Methodology

Configuration of Workflow Application

In Section 3.6.1, we extended XML structure of synthetic workflows to simulate stream workflow applications. Based on that, we model different stream workflow applications using common workflow structures (i.e. Montage, Inspiral, Epigenomics and Cybershake) for our experiments. Since each of these structures comes with different sizes, we can conduct small to medium to large experiments with different simulated workflow structures (i.e. stream workflows). Therefore, for each workflow structure, three different sizes of such structure are used (small, medium and large) as listed in Table 4.3.

Multicloud Environment

We model three cloud infrastructures (Amazon EC2 (Amazon, 2017a), Google Cloud Engine (Google, 2017), and Microsoft Azure (Microsoft, 2017)) with different VM configurations chosen from pre-defined machine types offered by those clouds. These VM configurations are

Table 4.3: Workflow structures with their different sizes

Size	Montage	Inspiral	Epigenomics	CyberShake
Small	25 node (Montage_25)	30 node (Inspiral_30)	24 node Epigenomics_24	30 node (CyberShake_30)
Medium	50 node (Montage_50)	50 node (Inspiral_50)	46 node Epigenomics_46	50 node (CyberShake_50)
Large	100 node (Montage_100)	100 node (Inspiral_100)	100 node (Epigenomics_100)	100 node (CyberShake_100)

provided in Table 4.4. We used the proposed IoTSim-Stream in Chapter 3 to simulate these infrastructures as a Multicloud environment. In CloudSim (Calheiros et al., 2011), MIPS rating is used to represent CPU unit, where the capacity of a VM instance is represented by the total MIPS assigned to such instance according to the assigned value of MIPS rating multiplied by the number of assigned CPU cores (Processing Elements (PEs) in CloudSim term). Hence, the processing power of each VM instance offered by the modelled cloud is converted to the corresponding MIPS value.

To convert the capacity of each VM instance offered by modelled clouds to corresponding MIPS value, we use the following approach: for Amazon EC2, CPU core provides the equivalent CPU capacity of 1000 MIPS [1] (1 ECU), for Google Compute Engine, CPU core provides the equivalent CPU capacity with 2750 MIPS [2] (2.75 ECUs), and for Microsoft Azure, CPU core provides the equivalent CPU capacity with 2500 MIPS [3] (2.5 ECUs).

Network Configuration

To model network performance (i.e. bandwidth and latency) of modelled clouds, we have conducted TCP bandwidth and latency tests between different zones of Nectar Cloud using IPerf (a cross-platform network performance measurement tool for both TCP and UDP) to collect the results for network bandwidth (in MB/s) and PING tool to collect the results for network latency (in second). From the obtained results, we create three ranges for bandwidth and latency for ingress and egress traffic as listed in Table 4.5 and 4.6 respectively.

Table 4.5: Ranges of ingress network bandwidth and latency.

Range	Minimum Bandwidth (MB/s) / Latency (seconds)	Maximum Bandwidth (MB/s) / Latency (seconds)
Low	302 / 0.0004	614 / 0.00063
Medium	615 / 0.00064	926 / 0.00086
High	927 / 0.00087	1238 / 0.0011

[1] For Amazon EC2, one ECU provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor, approximately 1000 MIPS (Javadi et al., 2011).

[2] For Google Cloud Engine, the Google Compute Engine Unit (GCEU) is defined as a minimum processing unit, which is the equivalent one ECU (Ahuja and Kaza, 2015). The CPU core in Google Compute Engine provides minimum processing power equivalent to 2.75 GCEUs (2.75 ECUs), approximately 2750 MIPS (Ahuja and Kaza, 2015).

[3] For the D1-5 v2, D2-64 v3 and F series of machine types in Microsoft Azure, these instances are based 2.4 GHz Intel Xeon E5-2673 v3 (Haswell) processor, the 2.3 GHz Intel Xeon E5-2673 v4 (Broadwell) processor and the 2.4 GHz Intel Xeon E5-2673 v3 (Haswell) processor respectively (Microsoft, 2017). Based on that, we can assume a CPU core is roughly equivalent to 2.5 ECUs, approximately 2500 MIPS.

Table 4.4: VM configurations of modelled clouds

Cloud Provider	VM Type	vCPUs/ cores	ECUs	Total MIPS	Memory (GB)	Price (¢/second)
Amazon EC2 (Windows instances)	m4.large	2	6.5 (7)	7000	8	0.0054
	m4.xlarge	4	13	13000	16	0.0107
	m4.2xlarge	8	26	26000	32	0.0214
	m4.4xlarge	16	53.5 (54)	54000	64	0.0427
	m4.10xlarge	40	124.5 (125)	125000	160	0.1067
	m4.16xlarge	64	188	188000	256	0.1707
	c4.large	2	8	8000	3.75	0.0054
	c4.xlarge	4	16	16000	7.5	0.0107
	c4.2xlarge	8	31	31000	15	0.0213
	c4.4xlarge	16	62	62000	30	0.0426
	c4.8xlarge	36	132	132000	60	0.0859
Google Compute Engine (n1-series)	n1-standard-1	1	2.75	2750	3.75	0.0014
	n1-standard-2	2	5.5	5500	7.5	0.0027
	n1-standard-4	4	11	11000	15	0.0053
	n1-standard-8	8	22	22000	30	0.0106
	n1-standard-16	16	44	44000	60	0.0212
	n1-standard-32	32	88	88000	120	0.0423
	n1-standard-64	64	176	176000	240	0.0845
	n1-highcpu-2	2	5.5	5500	1.8	0.002
	n1-highcpu-4	4	11	11000	3.6	0.004
	n1-highcpu-8	8	22	22000	7.2	0.0079
	n1-highcpu-16	16	44	44000	14.4	0.0158
	n1-highcpu-32	32	88	88000	28.8	0.0316
	n1-highcpu-64	64	176	176000	57.8	0.0631
Microsoft Azure (Windows D and F-Series)	D1 v2	1	2.5	2500	3.58	0.0035
	D2 v2	2	5	5000	7	0.0069
	D3 v2	4	10	10000	14	0.0137
	D4 v2	8	20	20000	28	0.0274
	D5 v2	16	40	40000	56	0.052
	D2 v3	2	5	5000	8	0.0054
	D4 v3	4	10	10000	16	0.0107
	D8 v3	8	20	20000	32	0.0214
	D16 v3	16	40	40000	64	0.0427
	D32 v3	32	80	80000	128	0.0854
	D64 v3	64	160	160000	256	0.1707
	F1	1	2.5	2500	2	0.0027
	F2	2	5	5000	4	0.0054
	F4	4	10	10000	8	0.0107
	F8	8	20	20000	16	0.0213
	F16	16	40	40000	32	0.0426

Table 4.6: Ranges of egress network bandwidth and latency.

Range	Minimum Bandwidth (MB/s) / Latency (seconds)	Maximum Bandwidth (MB/s) / Latency (seconds)
Low	24 / 0.009	121 / 0.020
Medium	122 / 0.021	218 / 0.031
High	219 / 0.032	314 / 0.040

Data Transfer Cost

For Internet egress traffic, the cost/rate of data transfer for each modelled cloud is based on the monthly usage tier and the destination zone. To model the costs of data transfer (in cents/MB) for our experiments, we find the minimum and maximum data transfer costs between modelled clouds, and then use them to create three ranges (low, medium and high) as listed in Table 4.7. For ingress traffic, the cost is 0.

Table 4.7: Ranges of outbound data transfer cost for clouds

Range	Minimum (cents/MB)	Maximum (cents/MB)
Low	0.005	0.012
Medium	0.013	0.019
High	0.020	0.025

Data Rate of External Source

To model data rate of external sources (IoT devices such as sensor), we choose minimum and maximum data rate based on different data rates of various technologies/standards of IoT defined in (Postscapes, 2017), where the minimum is 0.0013 MB/s and maximum is 12.5 MB/s. From the chosen minimum and maximum, we create three different data rate ranges for our experiment, as listed in Table 4.8.

Table 4.8: Ranges of external source data rate

Range	Minimum (MB/s)	Maximum (MB/s)
Low	0.0013 (10.7 Kbps)	4.2 (33.6 Mbps)
Medium	4.3 (34.4 Mbps)	8.4 (67.2 Mbps)
High	8.5 (68Mbps)	12.5 (100Mbps)

Types of Service

Since each service of workflow application can be either movable or unmovable, there is a need to determine how many of those services are movable and how many of those services are unmovable. For unmovable services, we need to specify the placement cloud for each one of them. By considering workflow application as strict application, the number of movable services are low compared with the number of unmovable services. With more flexible and pervasive workflows, the number of movable services are high compared with low number of unmovable services, where there is a possibility for this type of workflow application to be all of its services are movable. Thus, by considering the different natures of strict workflow applications and highly flexible and pervasive workflow applications, we create three percentages ranges of movable services in workflow application and listed them in Table 4.9.

Table 4.9: Percentage ranges of movable services

Range	Minimum (%)	Maximum (%)
Low	0%	34%
Medium	35%	68%
High	69%	100%

Data Processing Requirement of Services

To model data processing requirement for services (simple or/and complex services), we create different ranges for data processing requirement as listed in Table 4.10, based on

the following specified minimum and maximum values: the minimum value for data processing requirement for a service is 20 MI/MB (representing data processing requirement for simple aggregation functions) and the maximum value for data processing requirement for a service is 4000 MI/MB (representing data processing requirement for complex aggregation functions).

Table 4.10: Ranges of service data processing requirement

Range	**Minimum (MI/MB)**	**Maximum (MI/MB)**
Low	20	1347
Medium	1348	2674
High	2675	4000

Output Data Rate of Service

As the output data rate of a service is calculated using Equation 5.5, specification of the proportion of data generated by a service based on input streams can be used to model output data rates for services in workflow applications. For modelling different ranges of service output data rate, we define the minimum and maximum output proportion/percentage generated by service based on input streams to be 0.01/1% and 1.5/150% respectively, and use them to create three ranges for service output data rate as listed in Table 4.11.

Table 4.11: Percentage ranges of service output data rate

Range	**Minimum (proportion / %)**	**Maximum (proportion / %)**
Low	0.01 / 1%	0.50 / 50%
Medium	0.51 / 51%	1.0 / 100%
High	1.01 / 101%	1.5 / 150%

Data Processing Rate for Minimum Stream Unit

In workflow applications, the data rates streaming from different sources (either external sources or other services) as inputs to service are varied. Thus, to process these streams using compute resources of such service, these streams should be divided into portions and then be scheduled on those resources for processing. To achieve that, we need to determine the smallest stream unit per second that will be processed by each provisioned compute resource, where compute resource can process multiple units per second according to its computing capacity per second (MIPS). By specifying minimum stream unit for the whole workflow application, the data processing rate (MB/s) for processing this unit can be determined to ensure that each provisioned compute resource at least processes one unit per second. For our experiment, we create three ranges for data processing rate of minimum stream unit as listed in Table 4.12.

Table 4.12: Ranges of data processing rate of minimum unit

Range	**Minimum (MB/s)**	**Maximum (MB/s)**
Low	0.2 (=1.6Mbps)	1.0
Medium	1.1	2.0
High	2.1	2.9

Genetic Algorithm Configuration

To produce results in GA, we configure its parameters as follows: population size and generation limit are 50, elitism is 1, and the probability for crossover, mutation and replacement operations are 0.8, 0.3 and 0.2 respectively.

Other Simulation Parameters

The other parameters including data processing rate (α_{S_n}) and incoming data mode towards a service from its parent service(s) as inputs are fixed for all scenarios, and their values are system-calculated rate and replica respectively. The simulation time for all experiments is 3 minutes (180s).

Experiments and Scenarios

To evaluate the efficiency of the proposed algorithms (Greedy and GA) in term of execution costs, and study their behaviours in term of computational time and end-to-end latency, two sets of experiments are conducted.

First set of experiments (execution cost comparison with lower bound and fair-share method): We compare the results of execution costs obtained from the proposed algorithms (Greedy and GA) for executing the 12 modelled workflow applications with lower bound under varying of seven parameters. These parameters are data rate of external source (P1), data processing requirement of service (P2), output data rate of service (P3), type of service (P4), network bandwidth and latency (P5), cost of data transfer (P6) and data processing rate of minimum stream unit (P7). Thus, seven experimental scenarios are considered in this evaluation as shown in Table 4.13, where in each scenario, the low, medium and high ranges of the variable parameter will be studied. In regards to lower bound, we have relaxed many constraints including services datacenter placement constraint, VM provisioning constraint (selecting the cheapest VM across all datacenter VM offers), data transfer cost (using a lower cost value from the studied range) and network bandwidth constraint (using a lower bandwidth from the studied range which leads to reduction in data transfer cost by transferring less data). Then for each service, the cheapest VM from the placement cloud of this service is provisioned as many as is required to achieve the specified data processing rate. After that, the total execution cost (provisioning cost + data transfer cost) is calculated using Equation 4.7 during the period of time T. In addition, we compare the proposed algorithms with default scheduling method used in Apache YARN and Mesos. Apache YARN

Table 4.13: Scenarios of our experimental study

Scenario	**Fixed Parameters***	**Variable Parameter**
Scenario 1	P2−P7	P1
Scenario 2	P1 and P3−P7	P2
Scenario 3	P1−P2 and P4−P7	P3
Scenario 4	P1−P3 and P5−P7	P4
Scenario 5	P1−P4 and P6−P7	P5
Scenario 6	P1−P5 and P7	P6
Scenario 7	P1−P6	P7
*The values of fixed parameters are obtained from their medium ranges		

uses default Fair scheduling method to equal share cluster resources between applications over time. Apache Mesos is a cluster manager, where the default scheduling decision used by the master process to determine how resources will be assigned to each framework is Dominant Resource Fairness algorithm; this algorithm is a fair sharing model to multiple resource types. Therefore, we have implemented fair-share scheduler (which provisions the same VM as many as is required to achieve the specified data processing rate for all services in a workflow). Then, we compare the results produced by this scheduler with the results from the proposed algorithms. In the aforementioned comparisons, we consider the results obtained from lower bound as the base values.

Second set of experiments (proposed algorithms comparison): We use the computational time and average end-to-end latency recorded from the aforementioned scenarios to study and compare behaviours of the proposed algorithms for executing different stream workflow applications.

4.5.2 Experimental Results and Discussion

For our experiments, we use the proposed IoTSim-Stream in Chapter 3. The experimental scenarios are simulated to evaluate and compare the proposed algorithms with lower bound as well as with each others. In regard to the experimental results of average end-to-end latency, these results are collected after the system warmed-up (i.e. at second 120) to study the delay when simulation system is under highest pressure. For GA, we run each experimental scenario ten times, and average results are obtained and used in representation of experimental results.

We have examined the results of all scenarios looking for those results that have little difference or have similar behaviour, and those with different behaviours. In regards to experimental results for execution cost comparison, we found that the results of Scenario 1 and 2 can be represented by the result of Scenario 2 as it expresses the highest values. Similarly, the results of Scenario 5 and 6 can be represented by the result of Scenario 6 as it expresses the highest vales. Therefore, the execution cost results of Scenario 2, 3, 4, 6 and 7 will be presented. Appendix B.1 provides those results for Scenario 1 and 5. Moreover and for the algorithm comparison using average end-to-end latency, we found that the end-to-end latency results of Scenario 1, 2, 5 and 6 have somewhat close behaviour with slight difference,

therefore the result of Scenario 2 can be used to represent their behaviours as it represents the highest values. Therefore, the average latency results of Scenario 2, 3, 4 and 7 will be presented. Appendix B.2 provides the average latency results of Scenario 1, 5 and 6.

Figure 4.7 to Figure 4.11 depict the experimental results for the relative difference (in percentage) that achieved by the proposed algorithms and fair-share algorithm in comparison to lower bound in term of execution cost. From the experimental results shown in these figures, our analysis and findings are summarised into three discussion points (DPs).

DP1 As we expected, the presented results in these figures showed that the proposed GA achieved lowest relative differences of execution cost in comparison to greedy algorithm and fair-share method. This is clear due to GA being efficient at searching large and complex spaces by rapidly traversing these spaces and finding several satisfying candidate solutions (i.e. resource selection solutions) to choose from by evolving over generations of candidate solutions. GA surpasses the greedy algorithm in term of cost reduction by finding the best resource provisioning and scheduling solution with minimal execution cost (from those satisfying solutions) for different modelled workflow applications. Moreover, the relative differences of execution cost obtained by the proposed GA are low in most cases, which makes this algorithm produces total execution cost results that are close to the results of the most relaxed lower bound. Of course, in some cases, there is still a difference because of the lower bound produced unachievable results. The reason for that is the proposed GA considers both costs of resource provisioning and data transfer for each candidate solution that being generated in comparison to greedy algorithm which finds a solution that reduces only resource provisioning cost and ignoring the contribution of data transfer cost and then based on that solution, the data transfer cost is calculated and added to provisioning cost making the execution cost.

DP2 In very few cases (such as high range in Scenario 2 with Inspiral_100 and low range in Scenario 4 with Inspiral_50) where the relative difference of execution cost between the proposed greedy and GA is slight, this little cost reduction is still reasonable and can be considered as an extra cost-saving when workflow application runs for several minutes, hours or even longer. For instance, with high range in Scenario 2 with Inspiral_100, the cost-saving of running this application for just a hour is $\approx$ ($10.44).

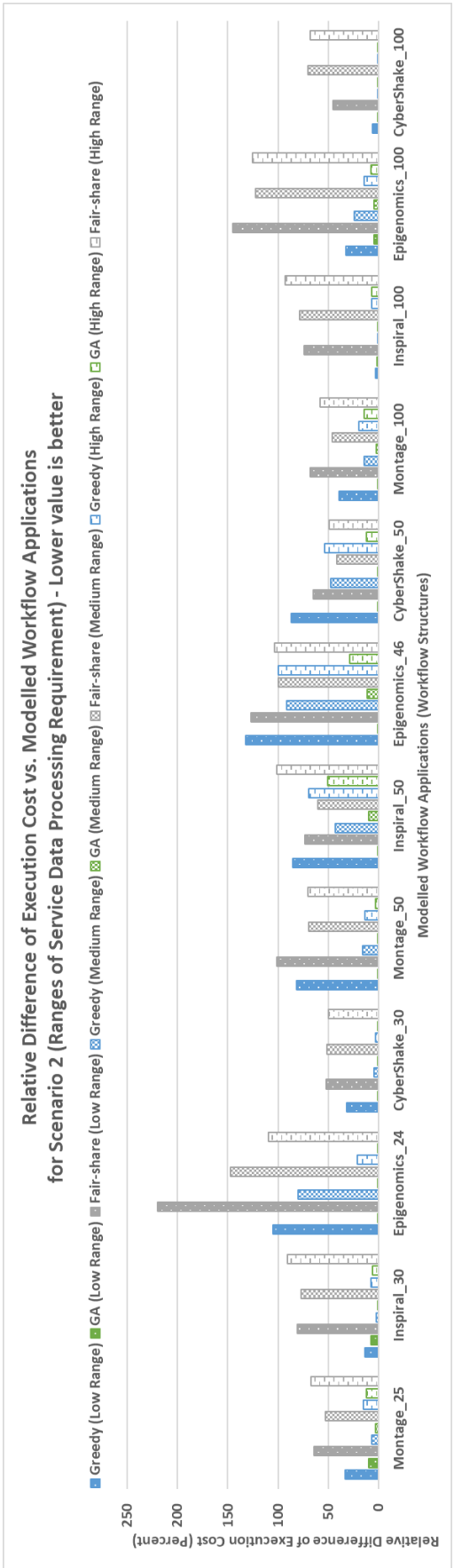


Figure 4.7: Execution Cost Comparison for Scenario 2.

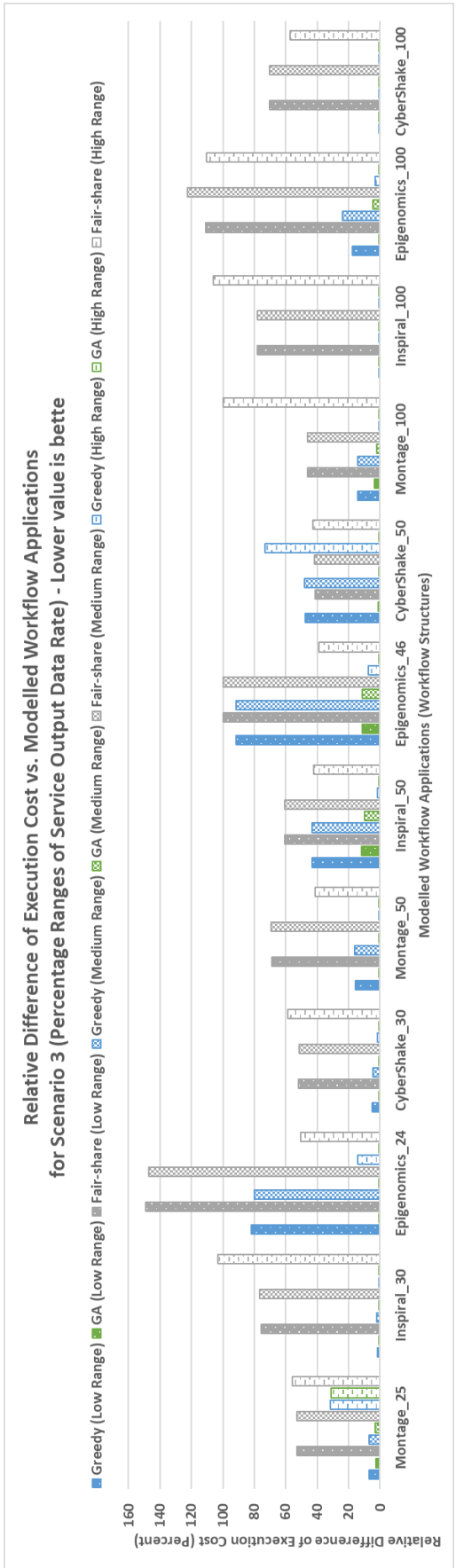


Figure 4.8: Execution Cost Comparison for Scenario 3.

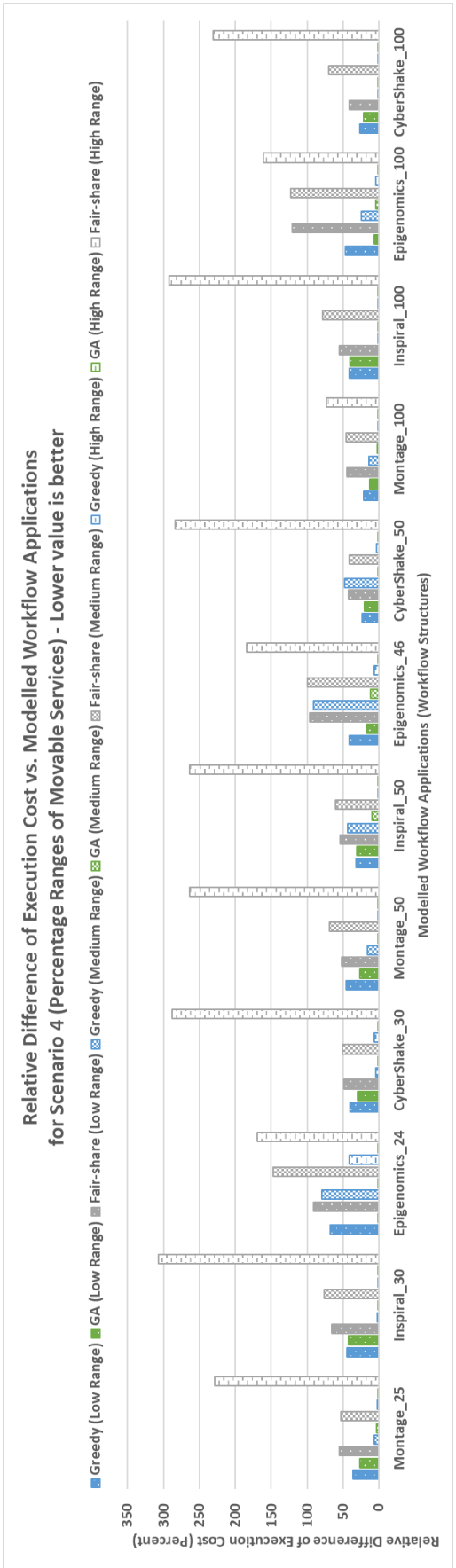


Figure 4.9: Execution Cost Comparison for Scenario 4.

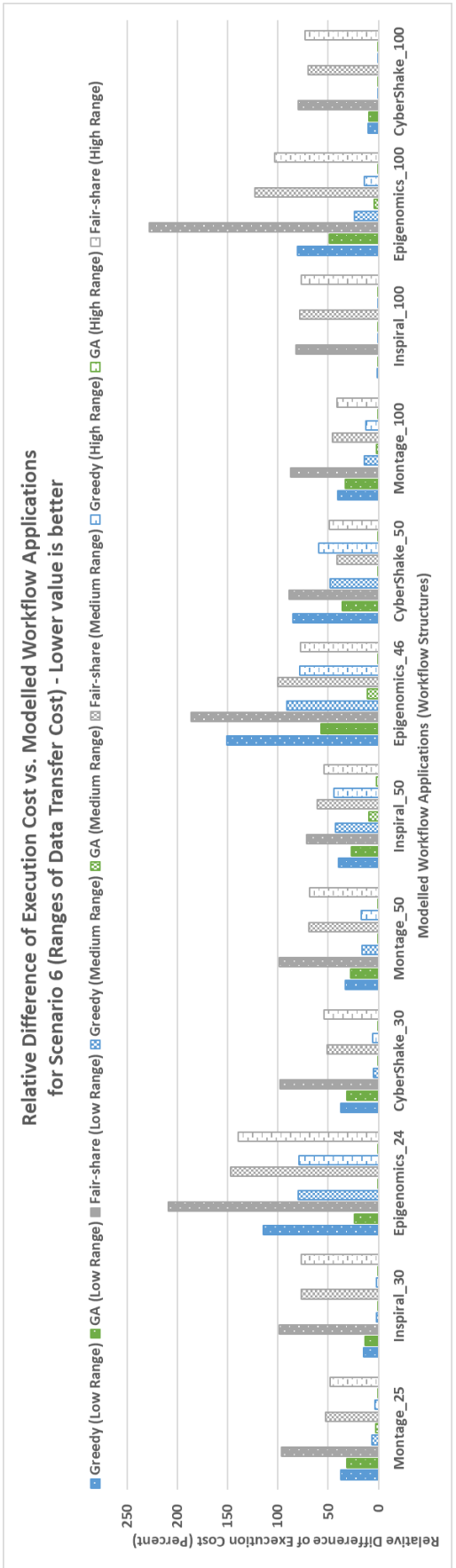


Figure 4.10: Execution Cost Comparison for Scenario 6.

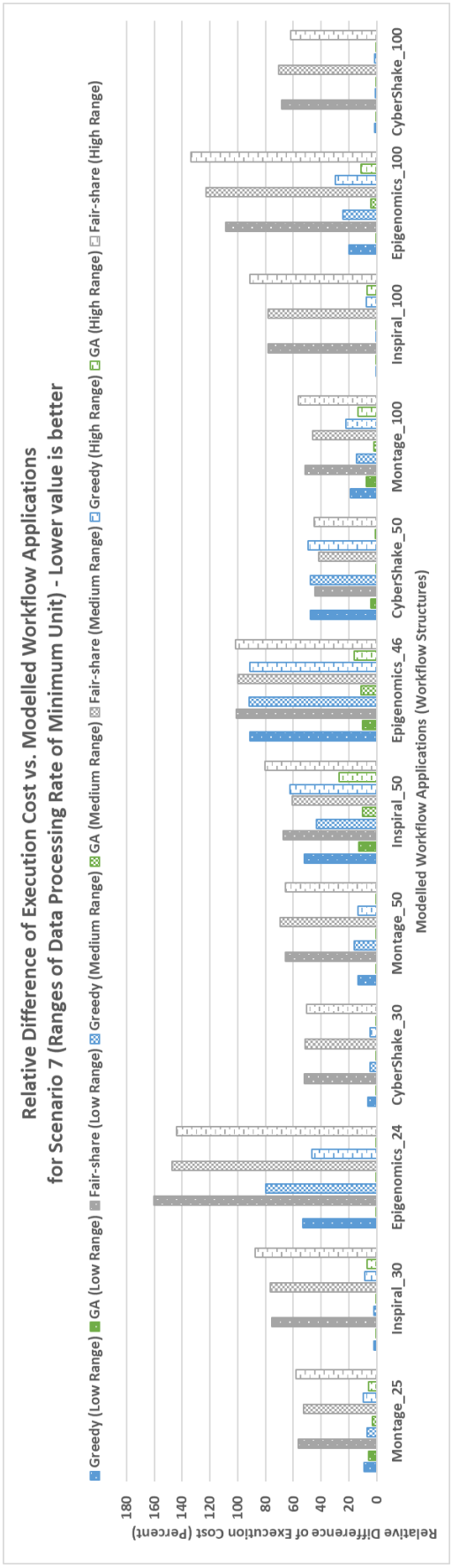


Figure 4.11: Execution Cost Comparison for Scenario 7.

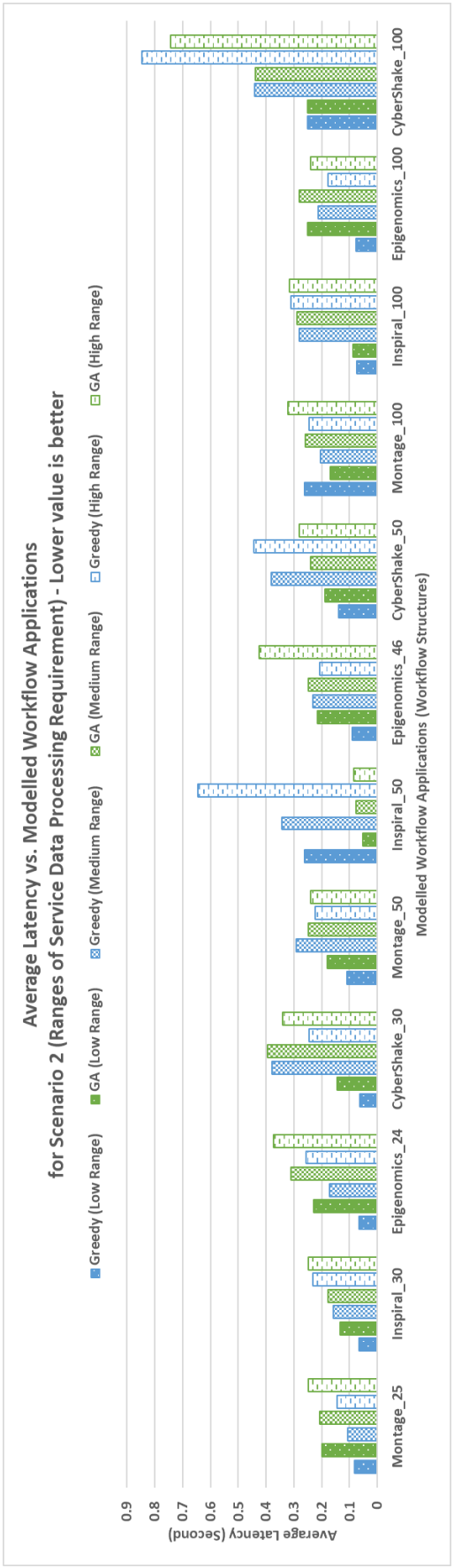


Figure 4.12: Proposed algorithms comparison using average end-to-end latency for Scenario 2.

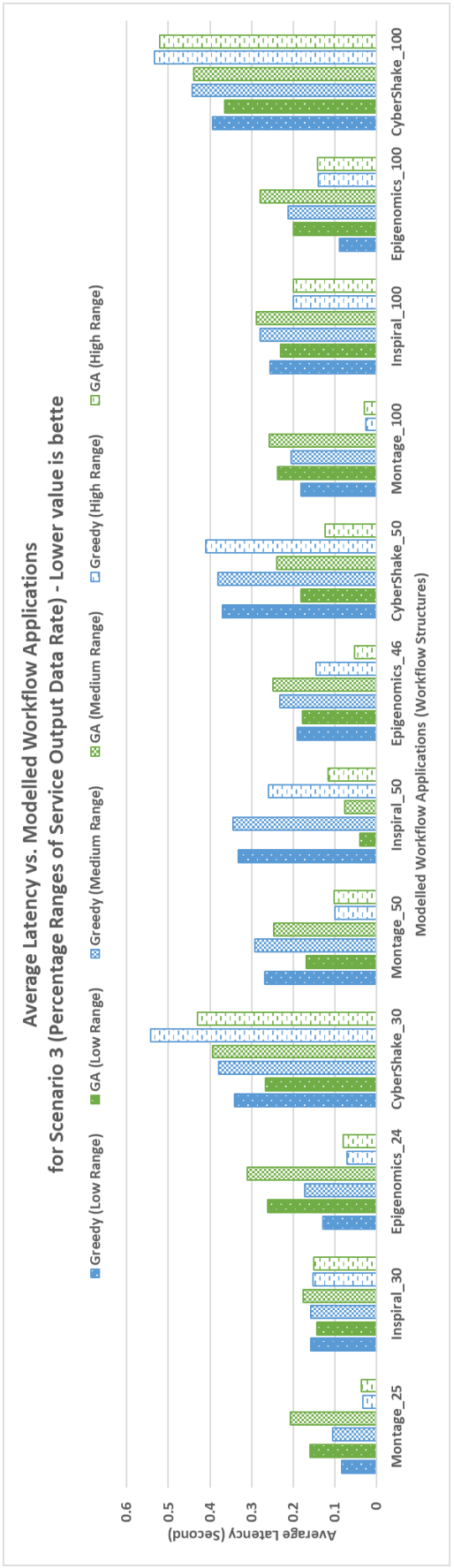


Figure 4.13: Proposed algorithms comparison using average end-to-end latency for Scenario 3.

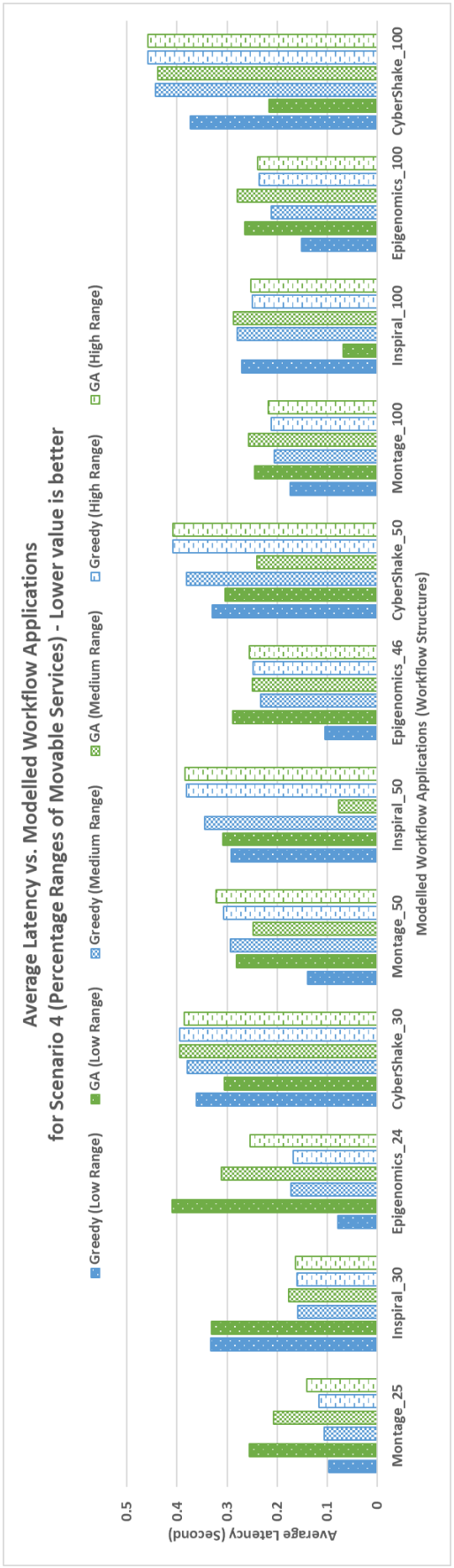


Figure 4.14: Proposed algorithms comparison using average end-to-end latency for Scenario 4.

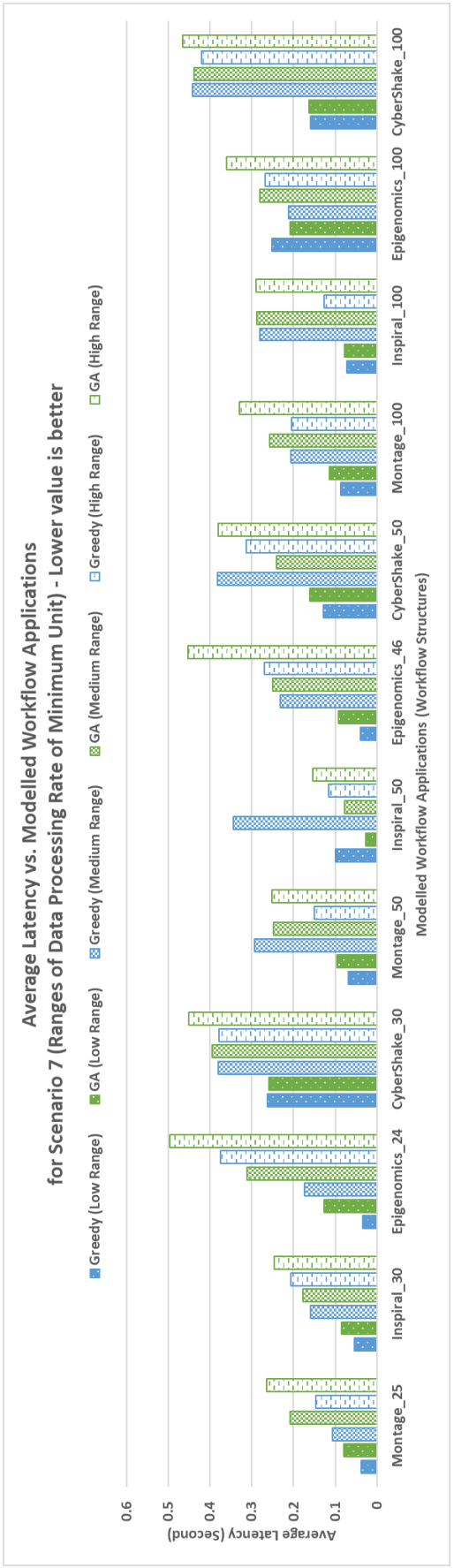


Figure 4.15: Proposed algorithms comparison using average end-to-end latency for Scenario 7.

DP3 By observing Figures 4.9 and 4.10, in some cases with low range, the relative difference of execution cost achieved by the proposed GA is not so close to unachievable lower bound. In relation to Scenario 4, the low percentage range of movable services means that there are high placement restrictions as most of services in workflow applications are unmovable, so that the opportunity of cost reduction is narrow and mainly based on the small number of movable services, leading to GA may not be able to find near-optimal solution for executing given workflow application. Whereas with high range, GA has an ample opportunity to find near-optimal provisioning and scheduling solution that leading to total execution cost results are closer to lower bound. In relation to Scenario 6, the reason behind that is when the cost of transferring data is low, GA may face a local optimality problem since changing the provisioning plan will not adjust the contribution of data transfer cost to the total cost as it is very low in origin.

For proposed algorithms comparison, the computational time results expressed the straightforward conclusion, which is the greedy algorithm takes less time to generate a scheduling plan compared with genetic algorithm, but we found that genetic algorithm needs relatively low time to compute and find such plan (at most across all scenarios). Therefore, we do not need to present these results and we only present the minimum and maximum computational time (in milliseconds) for each proposed algorithm with each scenario (see Table 4.14). In relation to average end-to-end latency, Figure 4.12 to Figure 4.15 show the average latency results achieved by these algorithms. Our analysis from these figures are summarised into three DPs:

DP4 It is clear that both algorithms are able to achieve sub-second average latency for 12 modelled workflow applications with all scenarios. The proposed greedy algorithm in most cases achieved lower average latency compared with GA. The reason behind that greedy algorithm is more oriented to provision each VM that not only achieve processing the minimum stream unit based on service data processing requirement but also has compute power to process the number of minimum units that being received as input stream portions to the service. However, as mentioned earlier, GA maintains sub-second average latency across all scenarios. It even achieved lower average latency in some cases compared to greedy algorithm such as in Scenario 3 with Montage_50, Inspiral_50 and CyberShake_50.

DP5 With Inspiral_50 and CyberShake_50 in Figure 4.12 and 4.13, the end-to-end latency of GA is lower than that of greedy algorithm. The reason behind that is the GA is designed to utilise data locality for all services within stream workflow. This minimises end-to-end latency by reducing data movement across multiple clouds and trying to avoid data transfer cost and time. For some cases, it is not applicable to achieve data locality due to several constrains such as huge number of data sources and their fixed placements.

Table 4.14: Computational Time Results (in Milliseconds)

	Greedy		GA	
	Min	**Max**	**Min**	**Max**
Scenario 1	1	219	57.6	2383.9
Scenario 2	1	391	65.8	2910.8
Scenario 3	1	2453	64.1	20127.9
Scenario 4	1	219	65.8	2383.9
Scenario 5	1	219	65.8	2653.9
Scenario 6	1	219	65.8	3280.4
Scenario 7	1	219	65.5	2383.9
Median	1	219	65.8	2653.9

DP6 The proposed algorithms are able to achieve the maximum throughputs that defined by the owner of workflow without affecting end-to-end latency and keeping average latency in sub-second since every data stream arrives is processed as soon as the dependency is achieved. The variations in the measured average latency occur because of the structure of workflow and the data dependency relations among services that are presented in this structure.

From the overall discussion in both comparisons, we found that GA achieved the best execution cost reduction, inexpensive computational time and good average latency while greedy algorithm achieved expensive execution cost, very low computation time and low average latency. For real-time data processing applications, end users are mainly concerned about the latency, but the expensive execution cost for the application is believed to be a barrier. Because of this, applications process big data also need large computational power. By considering the trade-off between the benefits of reduction in total execution cost and maintaining low computational time and end-to-end latency, we think that it is reasonable and practical to have low execution cost with little difference in average latency (bounded by a second) and computational time (bounded by several seconds). Thus, we can claim that GA is the best choice for meeting user performance requirements at deployment time while maintaining efficient performance (maximum throughput and sub-second latency) with minimal execution cost.

4.6 Summary

In this chapter, we addressed the problem of determining near-optimal resource allocation and scheduling of stream workflow applications and proposed two resource provisioning and scheduling algorithms (greedy and genetic) for efficient execution of such workflows in Multicloud environments. We also simulated different stream workflows using common workflow structures to examine the efficiency of the proposed algorithms in simulation environments using IoTSim-Stream. The experimental results obtained from our experiments showed that the proposed algorithms reduced the execution cost with modelled workflow applications, maintained throughputs and achieved sub-second latency. They performed better than the

fair-share algorithm. The proposed GA outperformed the greedy algorithm for all experiment scenarios.

After addressing the problem of scheduling stream workflows at deployment time, we can bring one type of runtime changes to this workflow, which is the change in data velocity. This change is considered as a non-structural change as the fluctuation of data does not lead to any change in the structure of data pipeline. The next chapter will discus the dynamic scheduling problem of stream workflows in a Multicloud environment while adhering to user-defined real-time performance requirements and handling data velocity changes.

Chapter 5

Dynamic Scheduling to Handle Data Velocity Changes

This chapter investigates the problem of managing resources over time to handle the load of incoming data with varying speed, because increasing the velocity of data without enough computing power leads to data loss and then violating real-time data analysis requirements. It proposes an adaptive scheduling technique to efficiently schedule stream workflow applications in a Multicloud environment and handle the changes in data velocity at runtime in order to always meet real-time data analysis requirements with minimal execution cost. The experimental results showed that the proposed technique is close to the lower bound and effective for different experiment scenarios.

5.1 Introduction

As we discussed before, stream workflows can be complex and involve heterogeneity, multiple data sources and multiple outputs. The most frequent runtime change with this workflow is the fluctuation of the data stream with time, so that the velocity of data may increase and decrease in an unpredictable manner. Considering stream workflows are adaptive workflow applications that serve the current-extra and future demands of changing real-time analytical requirements at runtime to make faster and better decisions, the resources should be managed over time. In this scenario, handling the fluctuations of data at runtime while meeting user-defined performance requirements needs to be investigated.

To address the problem of supporting dynamic scheduling under the variations of data stream rates, we design a new adaptive scheduling technique. This technique revises the scheduling plan of the stream workflow application according to changes happening in the speed of data at runtime to always meet real-time analytical requirements with minimal execution cost. In other words, it is aimed at tackling data stream velocity fluctuations while maximising performance efficiency, and all of that at minimal monetary cost.

This chapter is structured as follows: Section 5.2 reviews the related works. The problem formulation is presented in Section 5.3. Section 5.4 presents the proposed scheduling technique whose performance is evaluated in Section 5.5. We end this chapter with the conclusion in Section 5.6.

5.2 Related Work

In this section, we present comparisons with related works from three perspectives, which are application, modelling and methods/techniques.

From an application perspective, there are batch-oriented big data workflow (MapReduce workflow) and stream-oriented big data workflow (stream workflow). The focus of previous studies (such as (Wang et al., 2009b), (Wang et al., 2014a), (Teng et al., 2013), (Wang and Shi, 2014), (Shu and Wu, 2017) and (Zeng et al., 2016) (Zeng et al., 2018)) were mostly on MapReduce workflows and their executions in cloud computing infrastructure.

From a modelling perspective, there are two stream processing models, which are the data-flow graph with micro-batch processing model (i.e. discretised streaming model) and the operator graph with continuous processing model. With the discretised streaming model, streaming computations are performed on a series of small data batches called micro-batches. M. Zaharia et al. (Zaharia et al., 2012) followed this model and proposed a stream programming model named Discretized Streams (D-Streams). It brings together a series of Resilient Distributed Datasets (RDDs) and allows performing computations through various transformations. Apache Spark uses the RDD data model and allows to perform stream computations on RDDs to define data processing. While with the continuous processing model, an operator graph is used to model data pipeline, where each node in the graph is a

long-lived operator. This operator carries-out stream computation on streams as they arrive and produces new stream. Stream-oriented big data platforms and services such as Apache Storm and IBM Streaming allow to build streaming operator graphs for performing real-time data processing. As streaming operator graphs are different from dynamic stream workflows in that the source of data for the whole operator graph is one and there is one end operator, a new model is needed for dynamic stream workflow, which involves heterogeneity, multiple data sources and outputs.

From scheduling perspective, scheduling methods and techniques in the literature use heuristic and/or meta-heuristic approaches for making decisions based on different scheduling criteria (such as deadline, execution cost and performance) in order to meet user-defined SLA requirements. Research works such as D. Sun (Sun et al., 2015), T. Buddhika et al. (Buddhika et al., 2017) and A. Boek and F Werner (Bożek and Werner, 2018) focused on scheduling data stream computations for performance and/or energy optimisations. However, those research works and frameworks model stream workflow as a streaming operator graph. Since streaming operator graphs are different from dynamic stream workflows, the placement problem (i.e. scheduling problem) of dynamic stream workflows have different assumption and optimisation goals.

In the same perspective, D. Sun and R. Huang (Sun and Huang, 2016) and D. Sun et al. (Sun et al., 2018) focused on online scheduling with guaranteed makespan and utilised single cloud as an execution environment for big data streaming application. These scheduling strategies/methods do not consider stream workflow as a network of streaming big data workflow applications (i.e. workflow of workflows). They also do not take into consideration the dynamic nature of this workflow and its unpredictable performance, the various real-time decision support requirements and the powerful capability of 'cloud of clouds' as a dynamic execution environment. In the same context but for scheduling big data processing jobs/tasks and workflow in geo-distributed clouds, L. Chen et al. (Chen et al., 2018b) proposed fair job scheduler with the aim of reducing job completion time that relied on Apache Spark. Z. Hu et al. (Hu et al., 2016) proposed a new job scheduling method named Flutter, which aimed at reducing completion time and implemented in Apache Spark. H. Chen et al. (Chen et al., 2018a) proposed task-duplication based real-time scheduling method to reduce completion and execution times. However, these scheduling methods are considered stream workflow as operator graph and have different optimisation goals.

Accordingly, the scheduling techniques proposed in the aforementioned studies do not fit the composition needs of complex big data workflows. They also do not leverage the capability of Multicloud environment to cope with the dynamic aspects of these workflows. As a result, dynamic scheduling technique is needed for a stable and efficient execution of stream workflow over multiple cloud infrastructures that meets user real-time performance requirements and respond to the runtime changes in velocity of streaming data while reducing the overall execution cost.

Table 5.1: Problem Modelling Notation

Symbol / Term	**Description**
G	Workflow graph
S	Set of all graph services
EX	Set of all external sources
E	Set of all graph edges
e_m	Particular edge in workflow graph
¥^{m}	Percentage of data that is routed from parent service to child service (100% in replica mode or any percent in partition mode)
ψ^m	Data source on edge e_m that can be external source ex^m or origin service s^m_{dest} injecting its output data stream into the target of this edge s^m_{dest}
ex_p	Particular external source in workflow graph
Λ^{ex_p}	Output data rate of external source ex_p
S_n	Particular service in workflow graph
MI^{S_n}	Number of floating-point operations required to process one MB of input data (MI/MB)
λ^{S_n}	Amount of data produced by a given external source(s) and being consumed by a service S_n (MB/s)
γ^{S_n}	Proportion of output data to input data for S_n
C	Set of all clouds in Multicloud environment
c_g	Particular cloud in Multicloud environment
L	Network latency matrix
B	Network bandwidth matrix
D	Data transfer cost matrix
VM^g	Set of all VMs in cloud g
vm^g_k	Particular VM k in cloud g
U^g	Set of all internal network links between VMs in cloud g
u^g_h	Particular internal link between vm^g_{org} and vm^g_{dest}
$MIPS_{vm^g_k}$	Rating of the capacity of VM k in cloud g
$¢_{vm^g_k}$	Provisioning cost of VM k in cloud g (cents/s)
$\varphi(S_n, vm^g_k)$	Data processing rate for S_n when mapped to vm^g_k
$\varphi'(S_n, pro(S_n))$	Total data processing rate for S_n when mapped to all VMs in $pro(S_n)$
minDPUnit	Minimum stream unit for the whole application (MB)
unitDPRate	Minimum stream processing rate based on minDPUnit for the whole application (MB/s)
ϑ^{S_n}	P minDPUnits based on percentage change from original data rate that being increased or decreased from input stream of service S_n
χ	Service unit data processing rate
ϱ	Amount of output data stream of a service considering network bandwidth and latency

5.3 Problem Modelling

In Chapter 1, we presented a real use case for stream workflow application. From this application, the most dynamic form that occurs frequently is changing the velocity of streaming data for services. This is because the smart city is dynamic environment and the speed of streaming data is changing greatly based on time or traffic alert. Accordingly, in this section, we represent our previous problem modelling in Chapter 4 and extend it to model data velocity change at runtime. The list of all terminologies that will be used in this model is presented in Table 5.1.

5.3.1 Application Model

Stream workflow application can be represented as a DAG with $G = (S, EX, E)$. S represents a set of N services $S = s_1, s_2, ..., s_N$, EX represents a set of P external sources $EX = ex_1, ex_2, ..., ex_P$ and E represents a set of M edges/links between external sources and services, and between services themselves $E = e_1, e_2, ..., e_M$. Each edge, e_m is represented as a tuple $(\psi^m, s^m_{dest}, ¥^m)$, where ψ^m denotes stream output source which is either ex^m denotes external source or s^m_{org} denotes origin service, s^m_{dest} denotes destination service and $¥^m$ denotes the percentage of data generated by ψ^m that is routed towards s^m_{dest}. Each particular external source ex_p is represented as a tuple $ex_p = (\Lambda^{ex_p})$, where Λ^{ex_p} denotes the output data rate (data velocity) of this data source.

Each particular service S_n, is represented as a tuple $S_n = (MI^{S_n}, \lambda^{S_n}, \gamma^{S_n})$, where MI^{S_n} denotes the number of floating-point operations required to process one MB of incoming data (service data processing requirement) in MI/MB, λ^{S_n} denotes the arrival rate of data streams generated by sources outside the application in MB/s (such as data streams generated by sensors) to be consumed by the service, and γ^{S_n} denotes the proportion of data generated by the service based on input streams.

Notice that, given the nature of stream workflow applications, it is possible that data generated by one service can be sent to one or more services, or can be split among different services. Thus, for service S_n, both parameters γ^{S_n} and $¥^m$ (in edges where such service is origin service) are necessary to define the whole application. Additionally, the minimum stream unit per second (denoted as unitDPRate) should be defined to process streams that coming at different speeds, where each VM can process one or more units based on its computing power.

5.3.2 System Model

The cloud system is modelled as a tuple W = (C, L, B, D). A set of G clouds in the Multicloud environment is denoted as $C = c_1, c_2, ..., c_G$. L, B, and D denote matrices containing respectively the latency (in seconds), the bandwidth (in MB/s), and the data transfer cost (in cents/MB or ¢/MB) between each of the pair of clouds in C.

Each cloud, c_g is represented as a tuple (VM^g, U^g), where $VM^g = vm^g_1, vm^g_2, ..., vm^g_K$ is a set of K virtual machines (compute resources) with different resource configurations deployed in c_g, and $U^g = u^g_1, u^g_2, ..., u^g_H, u^g_h = (vm^g_{org}, vm^g_{dest})$, a set of H links that are part of the data center network topology.

Each VM deployed in the cloud, vm^g_k, is represented as a tuple $(MIPS_{vm^g_k}, ¢_{vm^g_k})$, where $MIPS_{vm^g_k}$ denotes floating-point operations computed by this VM according to its compute capacity per second and $¢_{vm^g_k}$ denotes the cost of provisioning such VM (in cents per second).

The data processing rate for S_n if it is mapped to vm^g_k is denoted as $\varphi(S_n, vm^g_k)$ and is calculated by dividing VM computing power by service unit data processing rate and service

data processing requirement as follows:

$$\varphi(S_n, vm_k^g) = \frac{\lfloor MIPS_{vm_k^g}/\chi \rfloor * \chi}{MI^{S_n}} \text{ MB/s} \tag{5.1}$$

$$Where\ \chi = unitDPRate * MI^{S_n} and\ MIPS_{vm_k^g} \geq \chi$$

As S_n could be mapped to more than one VM to achieve user performance requirements, let $pro(S_n)$ be the set of VMs that are provisioned from one cloud for service S_n. The data processing rate for S_n if it is mapped to VMs in $pro(S_n)$ is denoted as $\varphi'(S_n, pro(S_n))$ and is calculating by summing data processing rates for S_n on all provisioned VMs as follows:

$$\varphi'(S_n, pro(S_n)) = \sum\nolimits_{v \in pro(S_n)} \varphi(S_n, v) \tag{5.2}$$

In stream workflow application, the calculation of data processing rate for each service S_n should be carried-out at runtime. This is because of the need to handle dynamic changes that result in varying the speed of input streams being injecting into this service. Thus, system should calculate this rate based on the updated input speed of a service after the occurrence of change request at runtime. Let $inStream(S_n)$ denotes the input stream of S_n and is the total rate of incoming data from external sources and internal sources (i.e. parent services) based on data modes used to route such streams toward this service:

$$\begin{aligned} inStream(S_n) = \lambda^{S_n} + \sum\nolimits_{e_m \in E | \psi^m = s_{org}^m \& s_{dest}^m = S_n} \\ (\gamma^{s_{org}^m} * \varphi'(s_{org}^m, pro(s_{org}^m))) * ¥^m \text{ MB/s} \\ Where\ \lambda^{S_n} = \sum\nolimits_{e_m \in E | \psi^m = ex^m \& s_{dest}^m = S_n} (\Lambda^{ex^m}) \end{aligned} \tag{5.3}$$

The following data processing constraint of S_n is maintained:

$$\varphi'(S_n, pro(S_n)) \geq inStream(S_n) \tag{5.4}$$

Each service S_n produces output stream as a result of computation. Let $outStream(S_n)$ denotes the output data stream for a service S_n and is calculated by multiplying the total input rate of S_n by output data proportion/percent as follows:

$$outStream(S_n) = \gamma^{S_n} * inStream(S_n) \tag{5.5}$$

The velocity of data for given external source may change at runtime, which leads to a direct impact either an increase or decrease on the velocity of data for each S_n connected to this source. This change makes $inStream(S_n)$ and $outStream(S_n)$ be updated by the amount of data that being increased or decreased. Also, this change affects not only those services, but also has a subsequent change (i.e indirect impact) on the velocity of data for child services which have dependency-link with those services. Therefore, it is worth to note

that the maximum number of velocity changes that can be sent at any instant of time is assumed to be one and such velocity change request (either increase or decrease request) is only happen via external source. Let ϑ^{S_n} denotes the amount of data stream (in MB/s) based on percentage change from original data rate that being increased or decreased to $inStream(S_n)$ as P minDPUnits. In case of data velocity decrease, ϑ^{S_n} should be $0 < \vartheta^{S_n} < inStream(S_n)$. The $inStream(S_n)$ will be updated by adding or subtracting P minDPUnits and $outStream(S_n)$ will be updated by multiplying the update total input rate of S_n by output data proportion/percent as follows:

$$\begin{aligned} inStream(S_n) &= inStream(S_n) \pm \vartheta^{S_n} \\ outStream(S_n) &= \gamma^{S_n} * inStream(S_n) \end{aligned} \tag{5.6}$$

As well as the decrease in velocity of data for S_n leads to lower computing needs for maintaining the above data processing constraint, so that VM(s) that is not required will be deprovisioned. This results in cost reduction while meeting user real-time data processing requirements. While the increase in velocity of data leads to more computing demands to maintain the above data processing constraint for this high data rate, so that exVM(s) will be provisioned. Let $exVM(S_n)$ be the set of new VMs that need to be provisioned from placement cloud of service S_n to cope with the increase speed of data streams, and $rmVM(S_n)$ be the set of VM(s) from $pro(S_n)$ for service S_n that will be terminated/deprovisioned in response to an decrease in the speed of data streams. Thus, pro(S_n) is updated periodically at runtime by provisioning new VM(s) in case of velocity increases or deprovisioning VM(s) from the existing ones to respond to velocity decreases as follows:

$$pro(S_n) = \begin{cases} pro(S_n) + exVM(S_n), \textit{if velocity incr.} \\ pro(S_n) - rmVM(S_n), \textit{if velocity decr.} \\ pro(S_n), \textit{otherwise (no change)} \end{cases} \tag{5.7}$$

As pro(S_n) is updated at runtime, the $\varphi'(S_n, pro(S_n))$ is also updated, reflecting the new data processing rate for S_n based on the updated pro(S_n).

Given the change in velocity of data that either increases or decreases data rate which leads to provision more VMs or deprovision existing VMs at runtime, the execution cost needs to be calculated frequently. For our problem here, the calculation basis for the total execution cost of stream workflow application is per second. If T is total time duration, for cost calculations it is divided into several one second intervals (i.e. $t_1, t_2, ..., t_I$).

Additionally, we assume that every data stream should be processed, as unprocessed data streams lead to incorrect results. We also assume that the order of stream portions should be maintained during the distributed among the corresponding compute resources. Based on these assumptions, we maintain user specific throughputs for all services and end-to-end latency (response time) as low as possible or even bounded when it is being increased. Thus,

the incoming data streams are processed as they arrive and the latency is maintained, which is a time from a stream being added to input queue until its emission from the service as output stream. Of course, in case of a child service receives two or more dependency streams from its parents services, the latency is from the time of the last stream being added to input queue until its emission from child service.

The cost of running VMs used by service S_n to process incoming streams per second t_i is denoted as ec(S_n) while the total cost of running all VMs used by all services to process incoming streams during period of time T is denoted as ExecCost(S,T). The ExecCost(S,T) is calculated by summing VM provisioning costs for all services for T time as follows:

$$ExecCost(S,T) = \sum_{t_i} \sum_{S_n} ec(S_n) \qquad \text{cents} \tag{5.8}$$

The ec(S_n) is calculated by totalling the costs of all VMs provisioned for S_n per second as follows:

$$ec(S_n) = \sum_{v \in pro(S_n)} ¢_v \qquad \text{cents} \tag{5.9}$$

The data transfer cost is based on the amount of data being moved, the cost of data transfer charged by cloud provider, and network bandwidth. In a dynamic workflow application, the velocity of data determines the speed of generation, processing and analysis of data, where both input and output data are moved among different clouds. As we mentioned before, the change in velocity of data affects the data transfer cost as increasing speed leads to an increase in the cost and vice versa, so that the cost calculation needs to be carried-out per second. Let $cts(S_n)$ denotes the cost of transferring streams for S_n (including input streams from other services) per second, and $CTStream(S,T)$ denotes the total data transfer cost for the amount of data being moved for all services during the period of time T. The $CTStream(S,T)$ is calculated by summing the costs of data transfer between services for T time as follows:

$$CTStream(S,T) = \sum_{t_i} \sum_{S_n} cts(S_n) \qquad \text{cents} \tag{5.10}$$

The cts(S_n) is calculated by totalling the costs of data transfer performed by S_n per second as follows::

$$cts(S_n) = \sum_{S_i \in parent(S_n)} c(S_i) \text{ cents} \tag{5.11}$$

$$c(S_i) = \begin{cases} 0, & \text{if } C_g(S_i) = C_g(S_n) \\ outStream'(S_i) & \\ *D(C_g(S_i), & \\ C_g(S_n)), & \text{otherwise} \end{cases}$$

$$outStream'(S_i) \begin{cases} outStream(S_i), & \text{if } \varrho \leq 1 \\ \frac{outStream(S_i) * ¥^m}{\varrho}, & \text{otherwise} \end{cases}$$

$$Where\ \varrho = \frac{outStream(S_i) * ¥^m}{B(C_g(S_i), C_g(S_n))} + L(C_g(S_i), C_g(S_n))$$

$$, \ and\ parent(S_n) is\ the\ set\ of\ parent\ services\ for\ service\ S_n$$

Overall, the objective function is to minimise the cost of executing the dynamic workflow without violating data dependences and real-time performance requirements while dealing with changes in speed of data at runtime:

$$minf(S,T) = ExecCost(S,T) + CTStream(S,T) \tag{5.12}$$

Eq. 5.12 is solved for minimisation to generate cost-efficient scheduling plan for the execution of stream workflows. Considering services' data processing requirements and the variety of resources offered by multiple clouds, each service can be mapped to more than one resource in order to maintain its data processing constraint based on input data rate (refer to Eq. 5.2 and Eq. 5.4). If we relax such mapping constraint thus each service is mapped only to one resource (i.e. $|pro(S_n)| = 1$), assuming that this resource is sufficient to meet service's data processing constraint (Eq. 5.4), this relaxed constraint makes the problem 0-1 assignment problem. In this problem, the assignment matrix M indicates that a service i is assigned to resource j if $m_{ij} = 1$. This problem is well-known NP-hard (Martello, 1981). Consequently, if we consider the mapping of a service to more than one resource without any relaxation now, our problem is even harder than 0-1 assignment problem, so it is NP-hard problem. Moreover, our problem belongs to NP because if a feasible resource allocation solution is given, this solution can be tested in polynomial time using Algorithm 5. Accordingly, our problem is NP-complete problem.

Algorithm 5 polynomial-time algorithm for checking the feasible solution

```
1: totalDPRate ← 0
2: for each service S_n in S do
3:     for each VM vm_k^g from prov(S_n) do
4:         totalDPRate = totalDPRate + φ(S_n, vm_k^g)
5:     end for
6:     if totalDPRate < inStream(S_n) then
7:         return false {this is not feasible solution}
8:     end if
9: end for
```

5.4 Proposed Adaptive Scheduling Technique

As we discussed in the previous section, our scheduling problem is NP-complete problem. Thus, the problem's search spaces are complex, with large sets of VM offerings provided

by various cloud infrastructures and many constraints that need to be fulfilled such as data dependencies, user-defined real-time performance, throughput and end-to-end latency. Indeed, the search space of finding candidate solution for efficient execution of stream workflow application rapidly increases with the size of problem. Furthermore, the fluctuation of data velocity overtime makes it necessary to re-explore the complex search space in order to find sub-optimal solution as quick as possible, where exhaustive search for the optimal solution is not feasible. Consequently, th goal is to find near-optimal solution in the complex search space and revise it as fast as possible to tackle the changes in data velocity overtime without violating data dependences and real-time performance requirements while minimising the total execution cost (Eq 5.12). As we cope with velocity change for this workflow application, the following are cases of changing in input stream rate of a service:

- The speed of output stream of external source connected to this service is either increased or decreased.
- The speed of output stream of parent service(s) connected to this service is either increased or decreased. This happen when the increase or decrease in the speed of stream propagated from parent services due to the increase or decrease in the speed of stream for connected external sources

From the aforementioned goal, we have two challenges: (1) explore large search space to find candidate solution at deployment time and (2) revise this solution quickly and precisely with each velocity change request that occurs at runtime to locate sub-optimal solution to respond to such request. For the first challenge, genetic algorithm is useful algorithm in exploring complex search space to enable the practical implementation of optimising problem; thus, the objective function of Eq. 5.12 can be considered as a fitness function of genetic algorithm. While for the second challenge, Greedy heuristic can be used to adopt deployment plan generated by genetic algorithm at runtime because it provides an immediate sub-optimal solution for tackling velocity change request as it needs a relatively small time to compute; thus, it can fulfil the need to make scheduling decision under time constraints, enabling the practical implementation of optimising objective function at a given point. Accordingly, we propose a new adaptive scheduling technique for dynamic stream workflows.

The proposed technique is a two-phase dynamic workflow scheduling technique that incorporates two advanced optimisation algorithms (i.e. random immigrants genetic algorithm in Phase 1 and two-level greedy algorithm in Phase 2) to effectively perform adaptive scheduling of stream workflow applications in Multicloud environment and intelligently response to changes happen at runtime (i.e. velocity changes) with minimal execution cost. The main design goal of this technique is to find the best placement plan for the services of given workflow application with minimal execution and data transfer costs and maintaining its efficiency after each adaptation to handle the velocity change requests. The flowchart of the proposed two-step scheduling technique is depicted in Figure 5.1. In the below paragraphs,

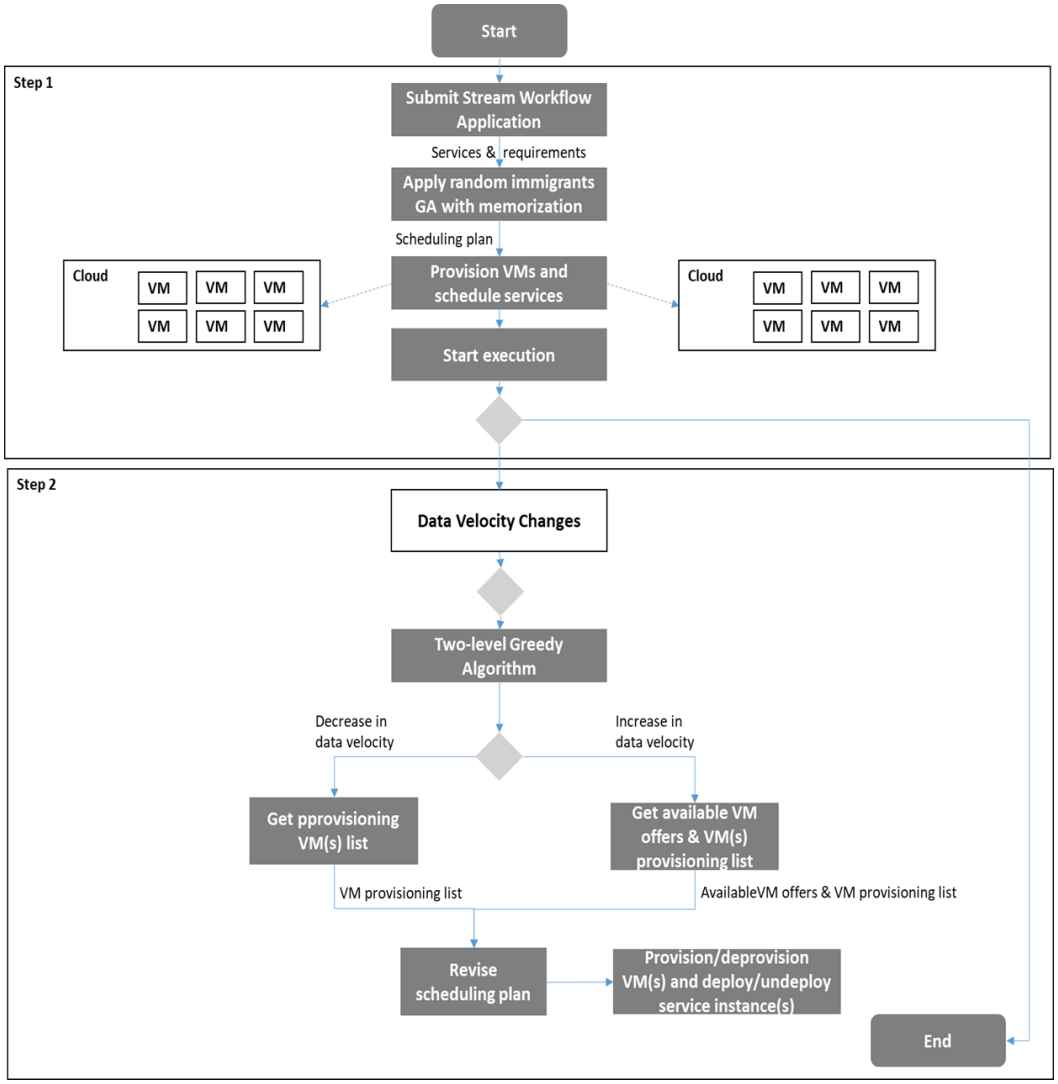


Figure 5.1: Proposed two-phase scheduling technique flowchart

we will discuss the two steps (and their sub-steps) of this technique. that exploiting the deployment flexibility provided by Multicloud environment.

First Step – The proposed random immigrants GA is called to find the best global sub-optimal resource selection solution according to the original real-time performance requirements to improve scheduling efficiency. Once the stream workflow application is scheduled on provisioned resources and being executed, the proposed technique is moved to the next step to tackle dynamic changes in the velocity of data streams.

Second Step – In this step, the proposed two-level greedy algorithm is used to dynamically respond to the changes in the speed of data streams for services. This algorithm at first level determines the services whose their input data rates will be changed due to the received velocity change request. Then in second level, it finds the best resource provisioning/deprovisioning solution(s) that will be used to tackle these changes and revising scheduling plan (to provision new VMs or deprovision existing ones that are not needed any more).

5.4.1 GA with Random Immigrants Scheme

Traditional GA has a considerable problem, which is convergence that prevents genetic diversity of the population. Therefore, to avoid such problem and to enhance the genetic diversity of the population, random immigrants schema is used (Yang, 2008). This schema retains diversity level of the population every generation via replacing a portion of candidate solutions in the current population with random candidate solutions called immigrants. Accordingly, we propose a random-based immigrants GA (GA for short) that is able to find sub-optimal resource selection solution for scheduling stream workflow application in Multicloud environment. It exploits data locality by selecting the most appropriate datacenter for each service, which leads to the reduction in both execution and data transfer costs. It generates the initial population randomly, and then evaluate the candidates and sort them in acceding order of fitness. During each generation, the elite candidate is selected and m random immigrants are generated then replaced the worst n candidates in the current population. Following the evaluation of m immigrants, the selection, crossover and mutation operators are applied. Finally, the elite candidate is added and the evolved population is evaluated and then sorted in acceding order of fitness before go to the next generation. Algorithm 6 shows the pseudocode of the proposed random immigrants GA provisioning and scheduling algorithm. The time complexity of this algorithm is presented in Table 5.2. The Watchmaker framework for evolutionary computation (Dyer, 2010) is used to implement this algorithm.

Table 5.2: Time complexity of random-based immigrants GA

Name	**Time complexity**
Random population generation	$O(su)$
Fitness Function	$O(ps^2d)$
Roulette wheel selection with binary search	$O(plog(p))$
Crossover	$O(s)$
Mutation	$O(sv)$
Sort	$O(plog(p))$
Random-based immigrants schema	$O(s^2d)$
Total	$O(gps^2d)$
g the number of generations (as termination condition), p the size of population, s the length of candidate solution (number of services), u the maximum number of required minimum data processing units of any service, v the number of VM offers in the placement cloud and d the maximum number of stream dependencies of any services	

5.4.2 Two-level Greedy Algorithm

We propose a new two-level greedy algorithm that uses Minimax with Alpha-Beta pruning method in game theory to minimise the maximum resource provisioning cost by finding the best resource selection solution for services that affected by data velocity changes. Minimax with Alpha-Beta pruning method is considered as a powerful searching and decision-making

Algorithm 6 GA with Random Immigrants Scheme

```
1: P ← empty initial population
2: generate N candidates randomly and add them to P
3: calculate fitness values for candidates in P
4: sort candidates in P in ascending order of fitness
5: while condition not satisfied do
6:     perform elitist selection
7:     P' = generate m random immigrants
8:     replace worst m candidates in P by random immigrants in P'
9:     calculate fitness values for random immigrants
10:    select candidates using selection operator for evolving
11:    create new offsprings using crossover operator
12:    create new offsprings using mutation operator
13:    add elite candidates to the evovled population
14:    calculate fitness values for candidates of the evovled population
15:    sort candidates of the evolved population in the ascending order of fitness
16: end while
17: return best candidate (candidate with minimum cost)
```

algorithm on game tree to find optimal/sub-optimal result from possible choices. Thus, this method is used in our algorithm to find the best resources with the lowest provisioning cost at runtime to achieve the updated data processing rate for each service affected directly and indirectly by velocity change request. The direct effect happens when the service is connected to external source whose data velocity will be changed, while indirect effect occurs when the service is in the velocity change path.

Our proposed algorithm addresses the problem of ongoing resource scaling under the dynamic variations of data stream rates by managing resources overtime. This algorithm at first level determines the services whose their input data rates will be changed due to the received velocity change request. Then, at second level, it finds the best resource provisioning/deprovisioning solution(s) that will be used to revise scheduling plan. With the occurrence of velocity change request, it finds the best provisioning and scheduling solution, and then dynamically and quickly updating the scheduling plan to respond to this change request while reducing the overall provisioning cost. The pseudocode of the proposed two-level greedy algorithm is shown in Algorithm 7 and the time complexity analysis of this algorithm is presented in Table 5.3. The pseudocode of two procedures that used in this algorithm to respond to velocity increase and decrease requests are shown in Algorithm 8 and Algorithm 9 respectively. The pseudocode of Minimax with Alpha and Beta algorithm that used in both procedures (Algorithm 8 and Algorithm 9) is shown in Algorithm 10. Algorithm 11 shows the pseudocode of evaluation function used in Algorithm 10.

Prior processing the velocity change request, the proposed technique finds the ids of service affected by this request directly or indirectly (Algorithm 7 Line 4). Then, for each service affected, it finds the best provisioning or deprovisioning solution based on the type of velocity change request. If the request is velocity increase request (Algorithm 7 Line 6), it calls Algorithm 8 to get VM offers of service placement cloud and then finds the extra minDPUnits that are achieved by the current provisioned VMs in accordance to service input data rate. Next, such algorithm calculates the number of minDPUnits required for data rate

Algorithm 7 Two-level Greedy Algorithm

1: $min \leftarrow TreeNode(-1, \infty)$ {-1 is vm global id and ∞ is value}
2: $max \leftarrow TreeNode(-1, -\infty)$ {-1 is vm global id and -∞ is value}
3: $depth \leftarrow 2$ {depth level in game tree}
4: *affectedSIDs* $\leftarrow$ get ids of services affected by velocity change request
5: **for** each service S_n in affectedSIDs **do**
6: **if** velocity change request is increase request **then**
7: Velocity_Increase_Req_Proc(S_n, min, max, depth)
8: **else**
9: Velocity_Decrease_Req_Proc(S_n, min, max, depth)
10: **end if**
11: **end for**

Algorithm 8 Velocity_Increase_Req_Proc(S_n,min,max,depth)

1: $reqUnits \leftarrow 0$
2: $unitMIPS \leftarrow MI^{S_n} * unitDPRate$
3: $avalVms \leftarrow$ get VM offers from service placement cloud
4: $avalVms = avalVms - \{x \in VM^g | MIPS_x < unitMIPS\}$
5: $extraAchievedUnits \leftarrow \varphi'(S_n, pro(S_n))/minDRRate - \lceil (inStream(S_n) * MI^{S_n})/unitMIPS \rceil$
6: $incRate \leftarrow$ get data rate increases over service input rate
7: $reqUnits \leftarrow$ get number of units required based on incRate
8: $reqUnits \leftarrow reqUnits - extraAchievedUnits$
9: $nodes \leftarrow$ create tree nodes for avalVms list
10: **while** $reqUnits > 0$ **do**
11: shuffle nodes and construct tree with specified depth
12: $root \leftarrow$ get root of constrcuted tree
13: $best \leftarrow$ Minimax_AlphaBeta(depth, true, root, min, max) {best node for VM selected}
14: $VMList = VMList \cup best.getVmgid()$
15: $reqUnits = reqUnits - \lfloor (MIPS_{best.getVm()}/unitMIPS) \rfloor$
16: **end while**
17: add VMList of S_n to ServiceVMsMap { VMList$\neq \phi$}

being increased over service input data and then deducts from this number the extra achieved units. After that, it calls Algorithm 10 several times to finds best VM(s) to provision until achieving the required units. While, with velocity decrease request (Algorithm 7 Line 8), it calls Algorithm 9 to get the list of VMs provisioned for a service and then finds the extra minDPUnits that are achieved by these VMs based on service input data rate. Next, such algorithm calculates the number of minDPUnits based on the data rate being decreased from service input data, and then increases this number by extra achieved units. After that, it removes those VMs from the list of provisioned VMs where their powers achieved units greater than the number of minDPunits that will be removed. The remaining VMs in this list will be used to find the best VM to deprovision using Algorithm 10.

Each run of the game finds the best VM to provision it in case of velocity increases or to deprovision it in case of velocity decreases. Since multiple VMs may be needed to achieve the updated data processing rate or may be released in response of decreasing the velocity,

Algorithm 9 Velocity_Decrease_Req_Proc(S_n,min,max,depth)

1: $redUnits \leftarrow 0$
2: $unitMIPS \leftarrow MI^{S_n} * unitDPRate$
3: SPVMs $\leftarrow pro(S_n)$
4: $extraAchievedUnits \leftarrow \varphi'(S_n, pro(S_n))/minDRRate - \lceil (inStream(S_n) * MI^{S_n})/unitMIPS \rceil$
5: $decRate \leftarrow$ get data rate decreases from service input rate
6: $redUnits \leftarrow$ get number of minDPUnits based on decRate
7: $redUnits \leftarrow redUnits + extraAchievedUnits$
8: **while** $redUnits > 0$ **do**
9: remove VM(s) from SPVMs achieved $units > redUnits$
10: **if** SPVMs is empty **then**
11: return {no provisioned VM can be deprovisioned}
12: **end if**
13: construct tree nodes from SPVMs list with specified depth
14: $root \leftarrow$ the root of constrcuted tree
15: $best \leftarrow$ Minimax_AlphaBeta(depth, true, root, min, max)
16: $VMList = VMList \cup best.getVmgid()$
17: $redUnits = redUnits - \lfloor (MIPS_{best.getVm()}/unitMIPS) \rfloor$
18: SPVMs = SPVMs − best.getVm()
19: **end while**
20: **if** VMList is not empty **then**
21: add VMList of S_n to ServiceVMsMap
22: **end if**

the game will be repeated to produce the best solution. For each VM selected, the number of minimum data processing units achieved based on the computing power of this VM in one game is deducted from the total required units (i.e. reqUnits) in case of velocity increases or from total reduced units (i.e. redUnits) in case of velocity decreases.

Table 5.3: Time complexity of two-level greedy algorithm

Name	**Time complexity**
Get affected services	$O(s)$
Velocity increase request procedure	$O(ub^m)$
Velocity decrease request procedure	$O(ub^m)$
Minimax alpha-beta	$O(b^m)$
Evaluation function	$O(1)$
Total	$O(sub^m)$
s the number of services, u the maximum number of required minimum data processing units of any service, b the branching factor and m the maximum depth of the tree	

Algorithm 10 Minimax_AlphaBeta(depth, maximisingPlayer, node, alpha, beta)

```
1: if depth == 0 then
2:    return evaluate(node)
3: else if maximisingPlayer then
4:    for each child of node  do
5:       TreeNode val =
      Minimax_AlphaBeta(depth - 1, false, child, alpha, beta)
6:       if val.getValue() > alpha.getValue() then
7:          alpha = val
8:       end if
9:       if beta.getValue() <= alpha.getValue() then
10:         break {alpha cut-off}
11:      end if
12:   end for
13:   return alpha
14: else
15:   for each child of node do
16:      TreeNode val =
      Minimax_AlphaBeta(depth - 1, true, child, alpha, beta)
17:      if val.getValue() ¡ beta.getValue() then
18:         beta = val
19:      end if
20:      if beta.getValue() ¡= alpha.getValue() then
21:         break {beta cut-off}
22:      end if
23:   end for
24:   return beta
25: end if
```

Algorithm 11 Evaluation Function - evaluate(node)

Require:

```
1: reqUnits, redUnits, unitMIPS
2: value, cost ← 0 {value for increase request and cost for decrease request}
3: if velocity change request is increase request then
4:    VMboottime ← get boottime for VM node
5:    achievedUnits ← get units achieved by VM node
6:    value ← (achievedUnits/(reqUnits ∗ ¢_{vm_k^g}))/VMboottime
7:    value ← value + ⌊MIPS_{vm_k^g}/(unitMIPS ∗ #OfServiceDependencies)⌋/¢_{vm_k^g}
8:    node.value ← value
9: else
10:   achievedUnits ← get units achieved by VM node
11:   cost ← (achievedUnits/(redUnits ∗ ¢_{vm_k^g}))
12:   node.value ← cost
13: end if
14: return node
```

5.5 Experiments and Discussion

5.5.1 Experiment Methodology

Configuration of Workflow Application

In Section 4.5.1, we simulated stream workflow applications using common workflow structures (Montage, Inspiral, Epigenomics and CyberShake) and described the additional parameter configurations. These applications with their different sizes are used in our experiments.

Multicloud Environment

In Section 4.5.1, we modelled three different cloud system providers, namely Amazon EC2 (Amazon, 2017a), Google Cloud Engine (Google, 2017), and Microsoft Azure (Microsoft, 2017), to form a Multicloud environment. This modelled Multicloud environment is used as an execution environment for our experiments.

In addition, to model boot time (startup time) for each VM configuration in the modelled clouds, we use average range of VM startup time defined in (Collins, 2015). For each modelled cloud, we generate random numbers from the defined range and then assign these numbers to its VM configurations.

Configuration of Data Velocity

To model the amount of data that is being increasing or decreasing in velocity change request for one external source, we utilised future data rates given in Gartner foreseen (Hassan et al., 2017b) which specifies one connected vehicle will generate as much as 25GB/hour of data, equivalent to 8MB/s. By considering this value as the average data rate of external source in workflow application, we create different percentage ranges for modelling the increase and decrease in data velocity. For velocity increase, we model the value of increase in data velocity as a percentage that is increased from current data rate. Similarly, we model the data velocity decrease as percentage of decrease in the current data rate. Table 5.4 lists the percentages of change to increase data velocity. Table 5.5 shows change percentages to decrease data velocity. It is worth to note that as there is a minimum limit for stream unit, the change value will be approximated/rounded to the nearest given minDPUnit. As an instance, if the minimum stream unit per second is 1MB/s and the 65% increase in data velocity from 5MB/s as original data rate is chosen randomly, the approximation will be applied on the change value (3.25MB/s) to be 3MB/s (i.e. the nearest value based on the specified minDPUnit) so that the new data rate will be 8MB/s.

Table 5.4: Percentage ranges of data velocity increase amount

Velocity Range	Minimum (Percent)	Maximum (Percent)
Low	10	30
Medium	50	70
High	90	100

Table 5.5: Percentage ranges of data velocity decrease amount

Velocity Range	Minimum (Percent)	Maximum (Percent)
Low	5	15
Medium	25	35
High	45	50

Workflow and Simulation Parameters

To run our experiments, we need to configure a set of parameters for both workflow application and simulator. These parameters and their values are fixed for all scenarios and listed in Table 5.6. For external data source rate, the value considered is from the data velocity configuration discussed in the previous subsection. For network bandwidth and latency for ingress and egress traffic, cost of data transfer and service data processing requirement, we considered the medium ranges of these parameters that presented in Section 4.5.1.

Experimental Scenarios

Our experimental evaluations for efficiency and performance of the proposed technique are described in the below paragraphs.

Comparison with baseline, GA and lower bound (Evaluation 1) − Study and compare the proposed adaptive scheduling technique (GA + two-level greedy algorithm) in finding the best resource provisioning solution and adapting scheduling plan in response to velocity increases/decreases with competitors (baseline algorithm and random-based immigrants GA scheme) and lower bound. This comparison is in term of the execution cost of different workflow applications for 3 minutes simulation time. A realistic baseline algorithm is created for our problem that does not need to use any complicated heuristic. It finds VM with the highest computing power and then provisioning it to respond to velocity increase requests, while with velocity decrease requests, it deprovisions one or more VMs from the available VMs to respond to these requests. The aim of comparison with baseline algorithm is to appreciate the necessity of our proposed technique to find the best resource provisioning solution and adapting the scheduling plan in response to velocity increases/decreases. The comparison with GA schema is aimed at evaluating the proposed technique with another meta-heuristic algorithm that is widely used in workflow scheduling research works in order to further proof its efficiency. Furthermore, the comparison with lower bound is to show that the complicated heuristic is necessary to approach the lower bound as well as to evaluate how the proposed technique is far from lower bound. In lower bound, we relaxed the same

Table 5.6: Workflow and simulation parameters

Parameter	**Value**
External Source Data Rate	5 MB/s with increase-velocity experiment 10 MB/s with decrease-velocity experiment
Ingress Network Bandwidth	Range [615, 926] MB/s
Ingress Network Latency	Range [0.00064, 0.00086] second
Egress Network Bandwidth	Range [122, 218] MB/s
Egress Network Latency	Range [0.021, 0.031] second
Data Transfer Cost	Ingress traffic: 0 Egress traffic: Range [0.013 - 0.019] cents/MB
Type of Service	50% unmovable services 50% movable services
Service Data Processing Requirement	Range [1348, 2674] MI/MB
Service Data Processing Rate	System-calculated rate based on input stream(s)
Data mode type	Replica
Service Output Data Rate	Range [1, 50] % of input rate
Minimum Data Processing Unit	1 MB
Minimum Data Processing Rate	1 MB/s
GA - Population Size	50
GA - Generation Limit	50
GA - Elitism	1
GA - Crossover Probability	0.8
GA - Mutation Probability	0.3
GA - Number of Random Immigrants	5
Number of Velocity Change Events	2
Delay between Velocity Change Events	10 seconds
Simulation Time	180 seconds (3 minutes)

constraints that were discussed in Section 4.5.1.

Guarantee processing speed for execution time (Evaluation 2) − Study the efficiency of the proposed adaptive scheduling technique in guaranteeing the processing speed required with different workflow applications under different velocity changes. This evaluation aims to show the data processing constraint is satisfying at all time with changing data velocity. The baseline here to achieve real-time user-defined requirements and end-to-end execution time is that the computing power available should be sufficient to process all incoming data without data loss. In other words, the computing capacity should be grater than or equal the velocity of incoming data at runtime. Thus, the sufficient computing capacity should be always maintained while the velocity of data increases or decreases at runtime. The experimental results will be collected before the end of simulation due to at that time all velocity changes have been made and handled by the proposed technique. This ensures the efficiency of the proposed technique to adopt scheduling plan in respond to velocity change request at runtime while guaranteeing processing speed required to achieve end-to-end execution time.

Efficiency of handling velocity change (Evaluation 3) − Study and compare the proposed adaptive scheduling technique with random-based immigrants GA scheme based on perform-

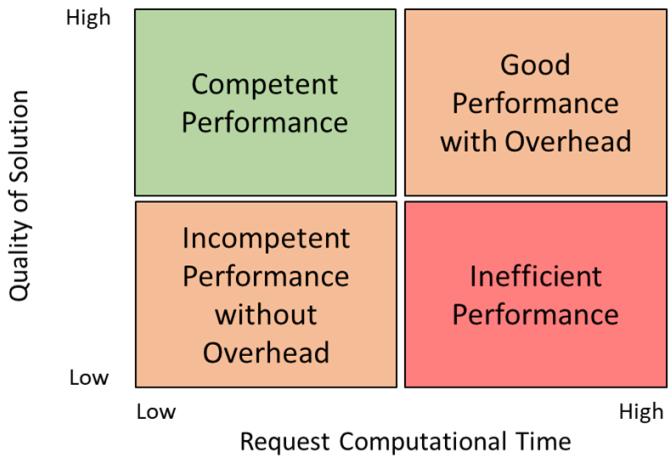


Figure 5.2: Performance Matrix

ance matrix presented in Figure 5.2 for performing dynamic scheduling at runtime. The aim of this evaluation is to determine how our proposed technique is effective in maintaining the quality of solution. This comparison is in term of the quality of solution for the revised scheduling plan, which includes solution cost (provisioning + data transfer cost per second) after the data velocity change request is applied, and the number of changes applied on the current scheduling plan to respond to this change request. The GA responses to each velocity change request by generating a totally new scheduling plan, which serves as a revised plan to replace the old one, while the proposed technique revises the current scheduling plan. When GA applied those VMs in a new plan that exist in old plan, these VMs are excluded to avoid VM duplication. By doing so, only VMs in the new plan that are not exist in the old plan will be provisioned and those VMs in old plan that are not part of the new plan will be deprovisioned. Therefore, the number of changes includes the changes in provisioning plan (for new VMs that are not in the old plan) and deprovisioning plan (for provisioned VMs that are not exist in the new plan).

In aforementioned scenarios, data rate of each external source in workflow application is set to be 5MB/s with velocity-increase experiment or 10 MB/s with velocity-decrease experiment at the beginning of execution. As velocity change requests being sent, the data rate of chosen external sources will be increased or decreased according to the conducted experiment.

By comparing the total execution costs of all workflow applications obtained from the proposed technique with the lower bound of total execution costs for these applications, we can evaluate the efficiency of this technique in finding the best solution either resource provisioning or deprovisioning solution in response to velocity increases or decreases. Also, by assessing the proposed technique's ability to guarantee data processing constraint all the time, we can further quantify its efficiency. In addition, comparing and evaluating the quality of solution generated by the proposed technique in comparison with GA allows to evaluate the performance of the proposed technique in relative to the performance of GA.

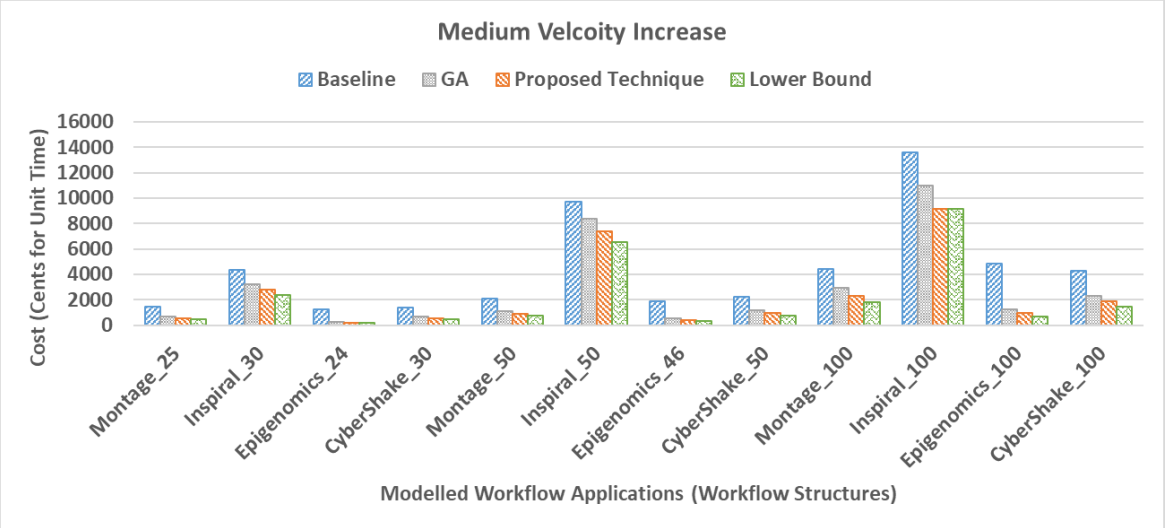


Figure 5.3: Total Execution Cost vs. Modelled workflow applications under medium range of velocity increase

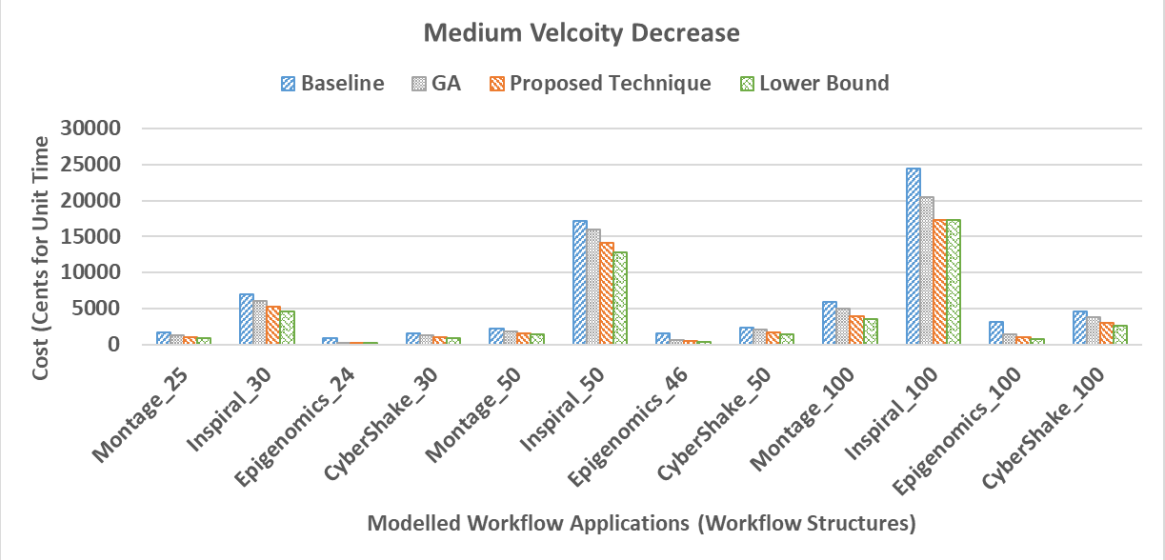


Figure 5.4: Total Execution Cost vs. Modelled workflow applications under medium range of velocity decrease

5.5.2 Experimental Results

To evaluate the efficiency and performance of the proposed technique, we conduct our experiments in simulation environment. This is because we need a controllable and repeatable environment to configure the parameters of each experiment scenario, and then compare the results obtained from the proposed technique with those from competitors under the same environment conditions. In real environment, some parameters like network bandwidth and latency cannot be controlled, making environment conditions are changing with each execution of workflow application. Thus, conducting our experiments in a real environment will produce inconsistent evaluation results, where these results cannot be used to assess the efficiency of proposed technique and the quality of solution produced to respond to data

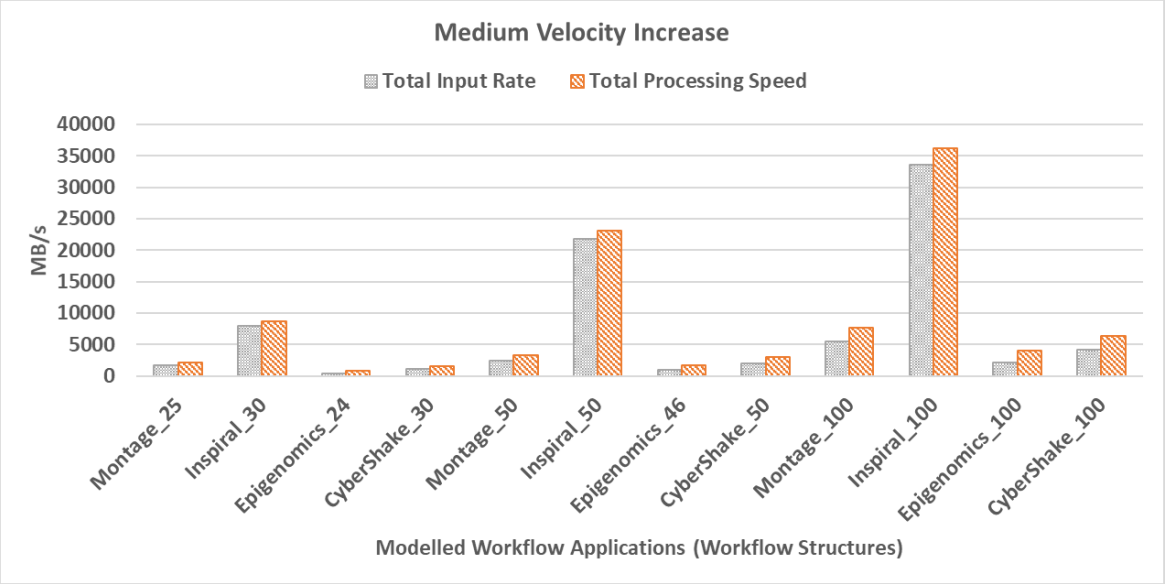


Figure 5.5: Total Input Rate vs. Total Processing Speed for different workflow structures (medium velocity increase range)

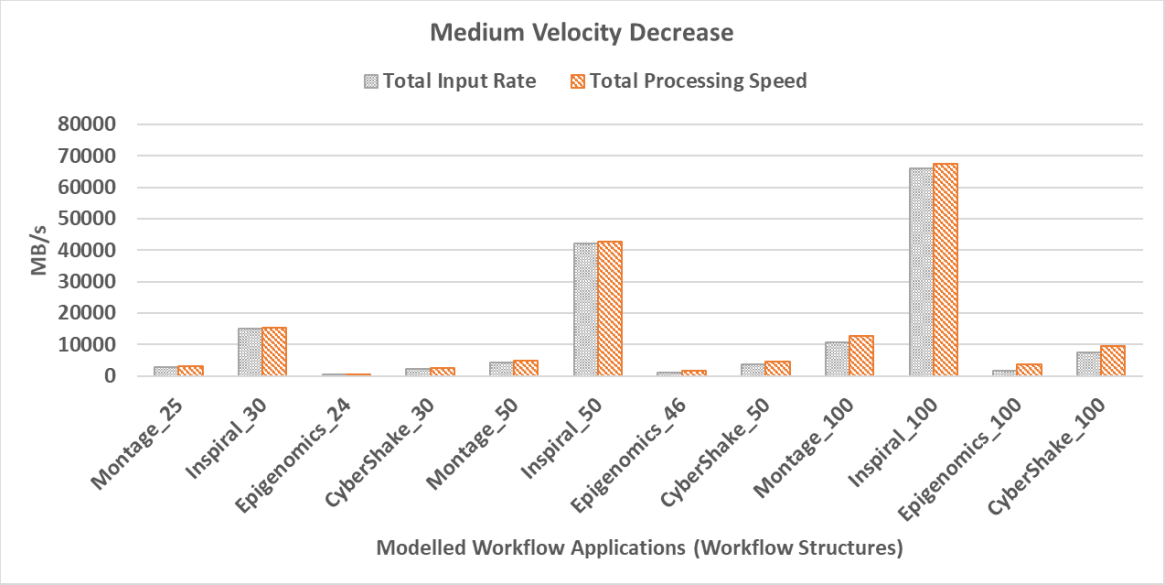


Figure 5.6: Total Input Rate vs. Total Processing Speed for different workflow structures (medium velocity decrease range)

velocity change requests at runtime. Accordingly, we conduct our experiments using the proposed IoTSim-Stream in Chapter 3.

The experimental scenarios are performed in simulation environment (by using IoTSim-Stream) on a Nectar Cloud virtual machine that had 8 vCPUs, 32GB of RAM memory and running Ubuntu 16.04.1 LTS, and the experimental results are collected. Since genetic algorithm is used in our proposed technique, each experimental scenario runs ten times, and the average value of the obtained results is taken and used in the representation of experimental results. Also, for the results of Evaluation 3, we present the average value for both solution cost and number of changes since two velocity changes are made during the

simulation time.

Evaluation 1: Results

We conducted experiments to record total execution cost achieved by proposed technique and the competitors (Baseline, GA and Lower Bound) for modelled workflow applications under different ranges of velocity increase and decrease. As the experimental results for the first evaluation showed that the total execution costs of modelled workflow structures under different velocity change ranges (low, medium and high) for both velocity-increase and velocity-decrease have not changed significantly, we only present those results for medium range of velocity change. The total execution cost results for low and high ranges of velocity changes are presented in Appendix C.1

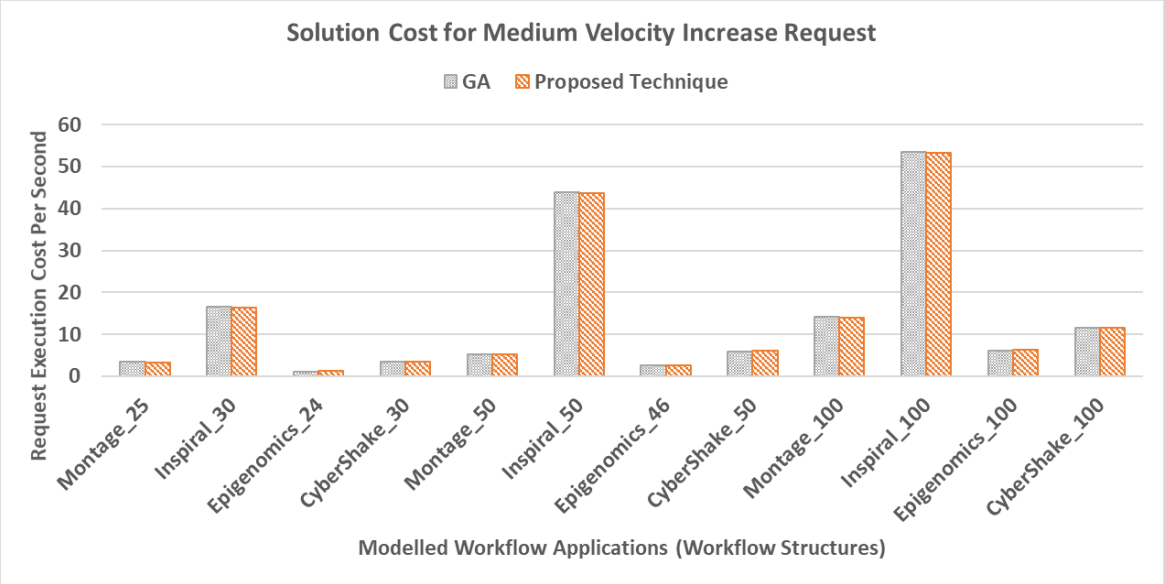
Figure 5.3 and Figure 5.4 depict the total execution costs of modelled workflow applications under medium range of velocity increase and decrease that achieved by baseline algorithm, GA, proposed technique and lower bound. From these results, our analysis and findings are:

- With various workflow applications, the proposed technique is efficient in finding the best solution to quickly respond to velocity change requests and then dynamically updating the current scheduling plan. The results of total execution cost obtained by the proposed technique compete the results obtained from both baseline and GA, and are close to the results of lower bound with most workflow structures. The reason behind that is the proposed technique uses GA at first phase for exploiting data locality to find near-optimal placement and scheduling plan, which reduces resource provisioning and data transfer costs, and then in the second phase, it uses greedy heuristic to find the best provisioning plan that reduces the provisioning cost as much as possible to respond to any velocity change request.

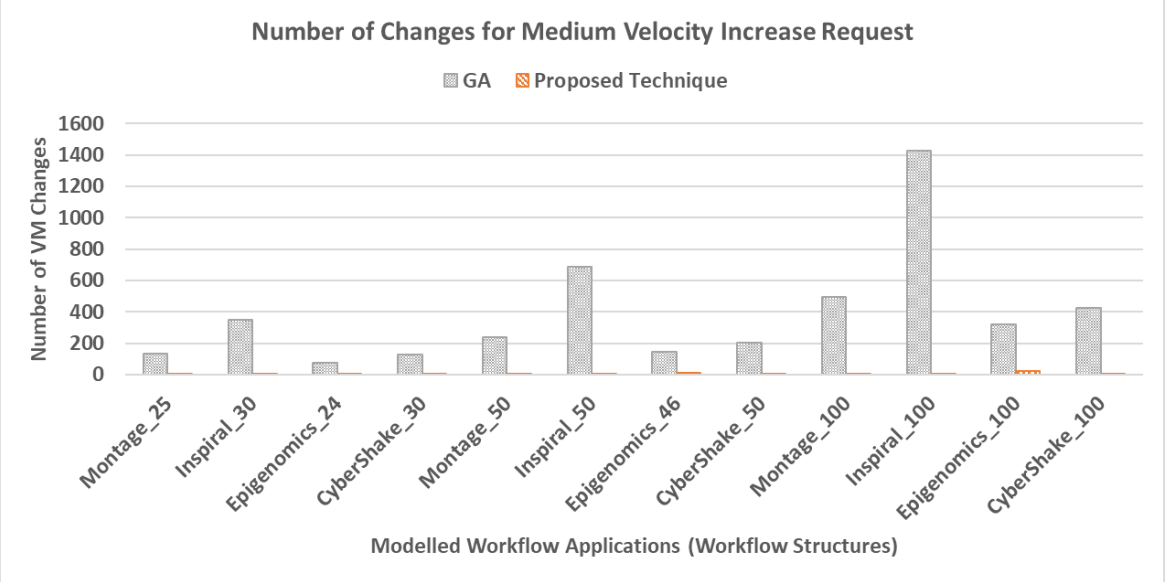
- The cost resulting from the proposed technique is a maximum of 32% of the cost generated by lower bound under medium velocity change. The reason for this difference is due to the structure of workflow may lead to process less data, so that the provisioning cost reduction factor contributes more to the total execution cost. Based on that, lower bound produces unachievable results as VM provisioning constraint is relaxed, while the proposed technique maintains this constraint.

- As data velocity increases from low to high range, the total execution cost for modelled workflow applications is slightly increased. The reason behind that is the proposed technique is able to revise the current plan to cope with velocity increase changes with minimal cost, leading to cost reduction even with high velocity of data.

- The proposed technique is an adequate and practical dynamic scheduling method with competent accuracy. This is because it takes all the defined constraints into consider-



(a) Solution cost



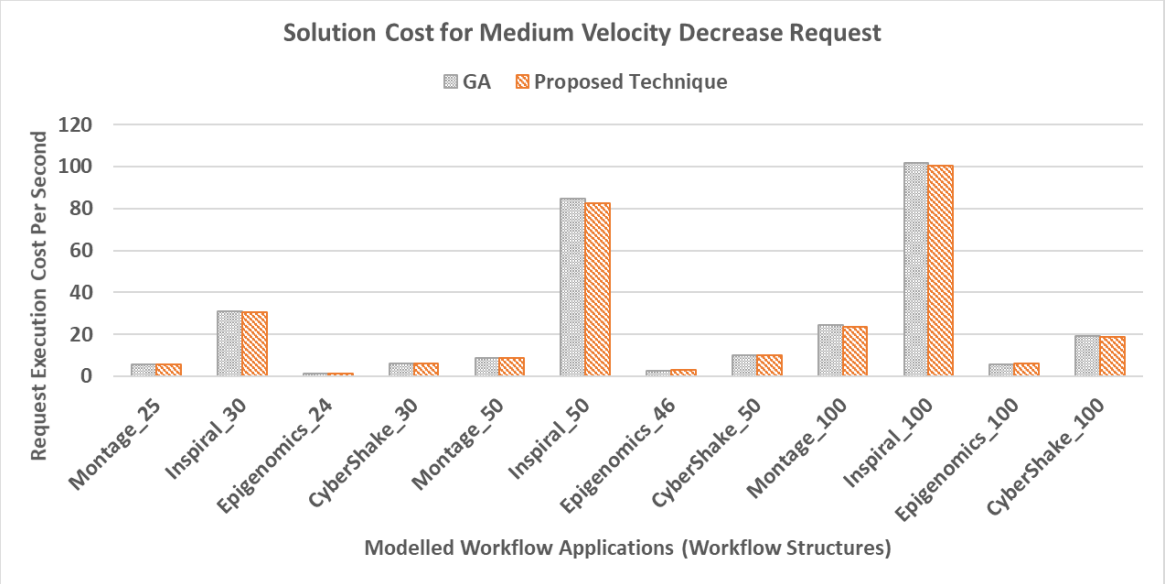
(b) Number of changes

Figure 5.7: Quality of solution for different workflow structures (medium velocity increase range)

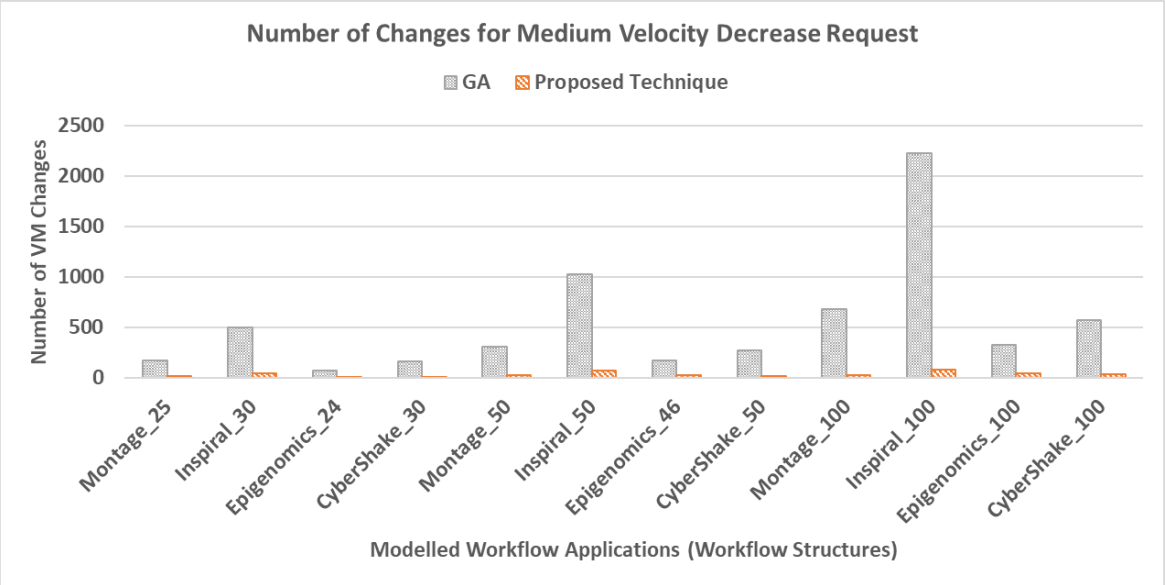
ation while meeting user real-time performance requirements and reducing the overall execution cost with different workflows.

Evaluation 2: Results

We conducted experiments to record total input data rate (in MB/s) and total processing speed (in MB/s) achieved by proposed technique for modelled workflow applications under different ranges of velocity increase and decrease. From the results obtained, we present here those results for medium range of velocity increase and decrease since these results are enough to reach to the conclusion. The total input rate and processing speed results for low



(a) Solution cost



(b) Number of changes

Figure 5.8: Quality of solution for different workflow structures (medium velocity decrease range)

and high ranges of velocity changes are presented in Appendix C.2. Figure 5.5 and Figure 5.6 show the experimental results achieved by the proposed technique in term of total input rate and total processing speed. From the presented results, it is clear that the proposed technique always guarantee processing speed to process incoming data with all workflow applications. Even more, it also has some extra computing power to handle increase in data velocity with immediate response and without the need to reschedule the execution plan.

Evaluation 3: Results

We conducted experiments to collect solution cost and number of changes achieved by proposed technique and GA for modelled workflow applications under different ranges of velocity increase and decrease. It is worth noting that we do not need to conduct experiments to record end-to-end latency, since our assumption in problem modelling is that every data stream that arrives will be processed as soon as the dependency is processed. Moreover, we do not need to conduct experiments to collect execution times required to respond to velocity change requests because the conclusion is straightforward. In regards to the computational time for velocity increase requests, the straightforward conclusion is that the GA needs more time to generate a new scheduling plan to respond to these requests, while the proposed technique quickly revises the current scheduling plan by relying on the two-level greedy algorithm for dynamic scheduling. In the favour of velocity decrease requests, the computational time needed for the proposed technique to respond is negligible since it just needs to deprovision the unnecessary VMs while the GA needs to generate a new scheduling plan, so that it is far-fetched for the GA to compete in that. We present the key important results from the second evaluation experiments, so the experimental results for the medium range of velocity changes (including both velocity increase and decrease requests) are provided. The quality of solution results for low and high ranges of velocity changes are presented in Appendix C.3. Figure 5.7 and Figure 5.8 show the experimental results achieved by the proposed technique in comparison to the GA in terms of solution cost and the number of changes required to revise the current scheduling plan. From the presented results, the following are our analysis and findings:

- In term of execution cost, GA with each request tries to find sub-optimal solutions by generating a new plan whilst the proposed technique just revises the current plan quickly, where the revised plan may not lead to sub-optimal solutions.
- In terms of the number of changes for a velocity request, the proposed technique responds to this request by quickly adjusting the current scheduling plan instead of generating a completely new scheduling plan, which usually requires a limited number of VM changes. In contrast, the GA generates a new scheduling plan in both velocity changes, which not only incurs more computational time but also requires a lot of VM changes to deprovision those VMs that are not in the new plan and to provision those that are in the new plan. To maintain the continuity of processing incoming streams at current data rates, unneeded VMs from the old plan must remain in use until the new VMs become ready. This causes further overhead in execution time and cost more as both new VMs and the current VMs (which will be deprovisioned later on) are remaining in resource pool. This also incurs more processing delays for upcoming streams when data velocity increases or more provisioning cost when data velocity decreases.

- From Figure 5.7b and Figure 5.8b, we can notice that the greatest performance gains are achieved by Inspiral_100. The reason behind that is the structure of this workflow processes huge amounts of data compared to the other workflows, resulting in higher computing power requirements. Thus, generating a new plan is too expensive and incurs a large number of changes, while revising the existing plan incurs a small number of changes that lead to huge performance gains.

- Based on the presented performance matrix (Figure 5.2), the proposed technique achieved competent performance with high quality of solutions with good execution cost compared to the GA with most workflows, a non-competitive number of changes are required to revise the scheduling plan, taking little or negligible execution time. Thus, the proposed technique outforms the GA with different workflow structures.

5.6 Summary

In this chapter, we considered the problem of dynamically scheduling stream workflow applications on various cloud infrastructures. These infrastructures form a Multicloud environment, which becomes the dynamic execution environment for these applications. To this end, we proposed a new dynamic scheduling and provisioning technique that incorporates a GA and a two-level greedy algorithm to efficiently schedule stream workflow application in Multicloud environment while meeting real-time user performance constraints under velocity changes with minimal execution cost. The experimental results showed that the proposed technique outperformed competitors in responding to data velocity changes at runtime while reducing the total execution cost for all modelled workflow applications under various data velocity ranges. It also close to the lower bound with a maximum of 32% execution cost.

After handling the fluctuations of data, we can bring runtime structural changes to stream workflow. These changes amend the structure of this workflow by adding new analytical components or modifying or removing the existing components. Thus, the next chapter will discus how to handle different structural changes of the stream workflow in a cost-effective manner while meeting user-defined real-time performance requirements.

Chapter 6

Dynamic Scheduling to Handle Application Structural Changes

This chapter investigates the problem of scheduling stream workflow to support runtime alterations of stream workflow deployment, so that the scheduling plan will be revised to handle stream workflow application with continuously changing characteristics. It proposes a pluggable dynamic scheduling technique that accepts user-defined algorithms to handle stream workflow runtime changes in a cost-effective manner in order to maintain real-time data analysis requirements. It also presents three different plug-in algorithms and methods to enable auto-scaling of stream workflows in a Multicloud environment. The experimental results of quality of solution showed that the proposed plug-in optimisation technique is more efficient to handle runtime changes compared to baseline technique and dynamic fair-share technique.

6.1 Introduction

With stream workflow, the fluctuation of data velocity is not the whole story of dynamism. This type of adaptive workflow becomes gradually more complex as its active analytical components can be adjusted over time according to changes in user-specific scenarios and the runtime environment. For instance from the real use case presented in Chapter 1, when vehicle environment or traffic conditions being changed in a smart traffic control service, a new analytical component may be added to a currently executing workflow or an existing analytical component might be changed or deleted to respond to the changes. Therefore, it is clear that these kinds of runtime changes are causing structural amendments in workflow application and even changes in velocity of data.

The focus of existing research works in the literature (such as (Liu et al., 2016b), (Liu and Buyya, 2017), (Kombi et al., 2019), (Sun and Huang, 2016) and (Sun et al., 2018)) is on streaming operator graphs and they mainly handle the fluctuation of data stream velocity and adjust resources to meet the needs of data processing. Also, some of these works consider the availability of computational resources or guaranteeing makespan. However, the current research problem is to serve the dynamic analytical components involved in a stream workflow, and delivering dynamic scaling and efficient performance under non-structural and structural changes of this type of workflow.

To address the aforementioned research problem, we investigate the problem of dynamically scheduling stream workflow to tackle its dynamic aspects at runtime, so that the stability of this application is maintained and achieved over time. We propose a fully-pluggable dynamic scheduling and provisioning technique that supports dynamic scaling by managing computing resources at runtime to respond to structural and non-structural changes of stream workflows in order to maintain real-time data analysis requirements. It amends the current scheduling plan according to runtime changes by using the plugged-in algorithms and methods that users defined by them to always maintain application stability. This proposed technique is considered as a future scheduling module in stream workflow management system, where its significance come from the flexibility provided to handle both non-structural and structural changes of running workflows.

This chapter is structured as follows: Section 6.2 presents a real use case for dynamic stream workflow application, while in Section 6.3, we present a structural change model for this type of workflow application. Section 6.4 explains in detail the proposed dynamic pluggable scheduling technique. Section 6.5 presents our experiment setup to evaluate the quality of solutions generated by three proposed plug-in scheduling techniques, while in Section 6.6, we discuss the obtained results to find the most efficient technique. Section 6.7 concludes the chapter and highlights future improvements.

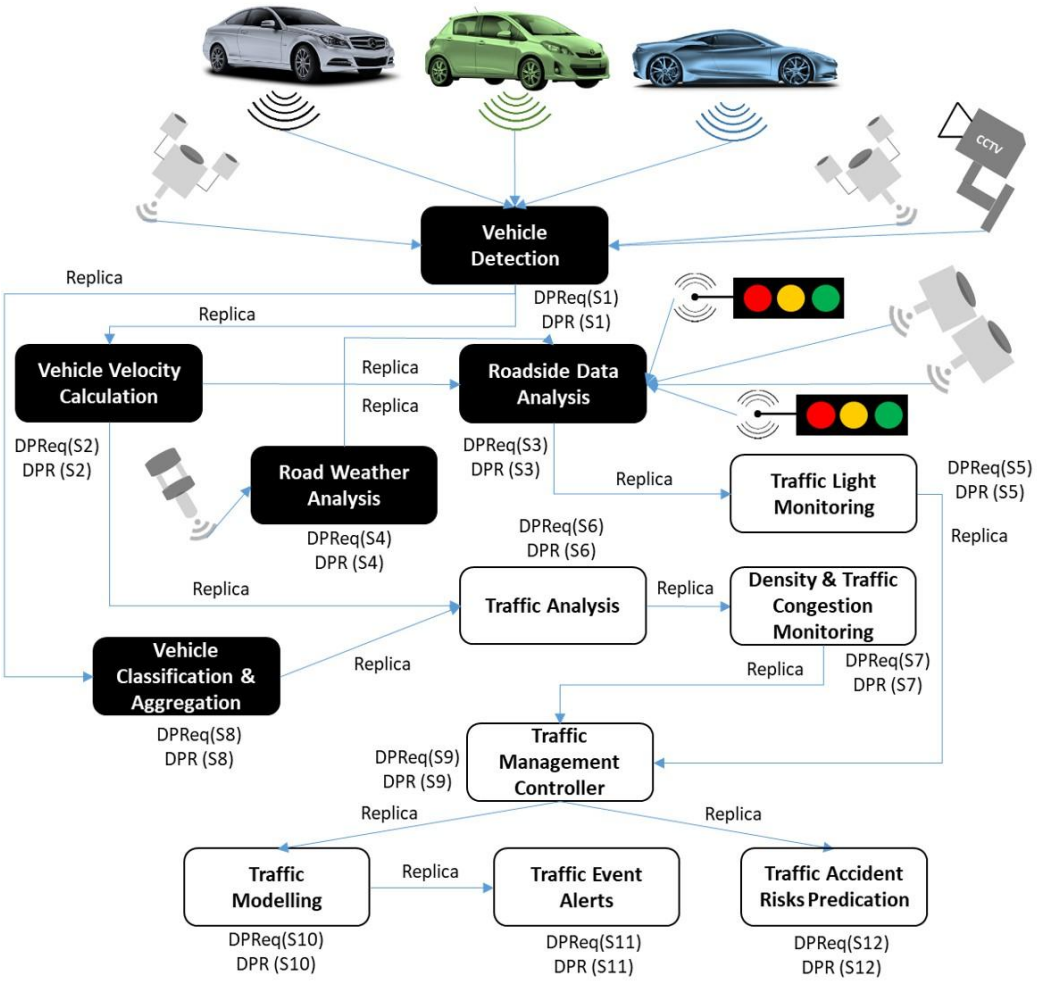


Figure 6.1: exemplar workflow for real-time view of road traffic and incidents

6.2 Real Use Case

In Chapter 1, we presented a real use case for stream workflow, which is connected vehicles in smart cities: real-time view of road traffic and incidents. The representation of the exemplar workflow of this real use case is shown Figure 6.1, which is more comprehensive than the one presented in Chapter 1. This workflow helps to enhance traffic flow and safety as well as it recommends the runtime adjustments based on live traffic events and road conditions (e.g. average running speed, traffic density (Chen et al., 2017)). The description and requirements of stream workflow can be found in Chapter 4.

This instant feedback use case shows the growing importance and value of real-time analytical insights in the future of smart city services (here road traffic monitoring service as real-world application). Such service application is a real-world dynamic big data pipeline that uses sensor data from connected vehicles, uploads such data to cloud datacenters for analysis, and produce real-time view of road traffic as continuous insights. From this application use case, we can outline the following dynamic forms:

- Dynamic environment of smart city − The incoming load for the analysis of streaming

data is varying greatly according to the number of connected vehicles available and being utilised (i.e. active connected vehicles).

- Velocity of Streaming Data − As smart city is dynamic environment, the speed of streaming data is changing greatly based on time or traffic alert. As an instance for the former variation, during peak hour traffic, a large number of connected vehicles operate on the road and they transmit their streaming data, whilst at night hour traffic, few vehicles operate (Chen et al., 2017). With the latter, light snow weather alert for example reduces the volume of traffic significantly (Akin et al., 2011), leading to transmit less amount of streaming data. This change in the speed of streaming data induces to different requirements of computing resources to cope with various incoming loads, so that the load (in term of data rates for components) at given time determines the required resources for computing. Therefore, the dynamic scaling and elastic of road traffic monitoring service application should be treated carefully for achieving real-time performance requirements under varying of data rates.

- Data processing requirement of analytical component− The real-time data processing requirement of analytical component may change overtime, reflecting the complexity of computations that will carried-out on data (from simple to complex aggregation functions and vice versa). This will affect the computing power needed according to the changes in data processing requirement.

- Structure of application − In smart city environment, the application analysis requirements change over time which reflect the new amendments to control and/or data flows. This means analytical components will be added and removed on the fly to achieve the new data analysis requirement. Thus, active connected vehicles and/or existing components may be connected to the components being added or their communications may be cut off from the components that will be deleted. According to the aforementioned dynamic changes, the structure of this application becomes dynamic. This means the requirements of resources will vary as application structure changes, where more computing resources are provisioned as long as new components being involved and they are deprovisioned as long as existing components being removed from data pipeline.

6.3 Problem Definition and Modelling

As stream workflow is an adaptive workflow application involving dynamic services, these services can be changed overtime depending on the amendments to control and/or data flows. These amendments are runtime changes that occur during the execution of this application. Tackling these changes is crucial. Thus, the dynamic scaling of stream workflow application should be treated carefully for handling these changes while achieving real-time performance

requirements. For our problem modelling here, we consider single service change at any instant of time. In other words, we assume that only one runtime change can be made at any instant of time and such change request requires one response action to cope with. Accordingly, and after examining the dynamic forms in the aforementioned real use case (Section 6.2), we address the following dynamic forms of stream workflow application:

- Change the velocity of streaming data (Dynamic Form 1) − It is a runtime action to change the velocity of streaming data, either increase or decrease as a consequence of the change happens via external source (i.e. data rate is changed) or parent service (i.e. the velocity of output data is changed). This change induces either increase or decrease in velocity of data.
- Change the existing service (Dynamic Form 2) − It is a runtime action to change the existing service by amending either its data processing requirements (Case 1) or the velocity of output stream (Case 2). These cases will be discussed in this section.
- Add a new service (Dynamic Form 3) − It is a runtime action to amend the structure of application by adding a new service. This form induces five cases that will be discussed in this section.
- Delete an existing service (Dynamic Form 4) − It is an runtime action to amend the structure of application by deleting an existing service. This form induces two cases that will be discussed in this section.

We present here a structural change model that is an extension to our problem modelling presented in Chapter 5.

6.3.1 Dynamic Form 1: Change the Streaming Data Velocity

In Chapter 5, we proposed two-phase adaptive scheduling technique to efficiently reschedule dynamic workflow application in cloud infrastructures to respond to changes in the velocity of data at runtime. The details of this technique and how it used to handle such dynamic form are provided in that chapter.

6.3.2 Dynamic Form 2: Change of Existing Service

Considering stream application as an adaptive workflow, any service (analytical component) may be changed overtime. This dynamic form occurs either in term of data processing requirement for a service MI^{S_n} or velocity of output data stream for a service $outStream(S_n)$ (i.e. as consequence of changing output proportion γ^{S_n}).

Case 1: Change data processing requirement of existing service. This change happens in real-world when a new version of existing service is available and be deployed in replace of the current version. For instance, in Figure 6.1, the new release of traffic analysis service is available. With this case, the user-supplied input and impact are as follows:

User Input: The service S_n selected, and the new value for MI^{S_n} denoted as nMI^{S_n}.
Impact: The MI^{S_n} and $pro(S_n)$ are updated as follow:

$$\begin{aligned} & MI^{S_n} = nMI^{S_n} \\ & pro(S_n) = \begin{cases} pro(S_n) + exVM(S_n), \textit{if } MI^{S_n} \textit{increases} \\ pro(S_n) - rmVM(S_n), \textit{otherwise} \end{cases} \\ & \textit{where } MIPS_v \geq unitDPRate * MI^{S_n} \mid v \in pro(S_n) \\ & \textit{and data processing constraint is maintained (Eq.5.4).} \end{aligned} \tag{6.1}$$

Case 2: Change output stream velocity of existing service. This change happens in real-world when a new release of service is deployed in replace of the current version, which produces less or more output data from the processed input data based on the new processing logic. For instance, in Figure 6.1, the new release of traffic analysis service is being deployed that changes the velocity of output stream either increases or decreases, so that downstream services (density & traffic congestion monitoring, traffic management controller, traffic modelling, traffic event alerts and traffic accident risks predication services) receive more or less streaming data. With this case, the following are user-supplied input and impact:
User Input: The service S_n selected, and the new value for γ^{S_n} denoted as $n\_\gamma^{S_n}$.
Impact: The γ^{S_n} and $pro(S_n)$ are updated as follow:

$$\begin{aligned} & \vartheta^{S_n} = \begin{cases} (n\_\gamma^{S_n} - \gamma^{S_n}) * inStream(S_n), \ \textit{if } \gamma^{S_n} \textit{increases} \\ (\gamma^{S_n} - n\_\gamma^{S_n}) * inStream(S_n), \ \textit{otherwise} \end{cases} \\ & \vartheta^{S_n} = \lceil \vartheta^{S_n} / minDPUnit \rceil * minDPUnit \\ & \gamma^{S_n} = n\_\gamma^{S_n} \\ & outStream(S_n) = \gamma^{S_n} * inStream(S_n) \\ & \textit{For each downstream service affected from this runtime change,} \\ & \textit{perform Eq.5.6 and Eq.5.7} \end{aligned} \tag{6.2}$$

6.3.3 Dynamic Form 3: Add a New Service

Adding a new service to stream workflow application at runtime means by the way performing a change in the structure of application. This change can be in various forms depending on where the new service is added in stream workflow, what is/are the input link(s) for such new service and where the output stream of this new service is routed. Considering these aspects, we support five different cases under this dynamic form, where each one of them has its own input requirements and impact. Nevertheless, the common impact of all cases is the creation of a new service S^* with its type, data processing requirement MI^{S^*}, placement cloud $c_g^{S^*}$ and output data proportion γ^{S^*}, and the update of S set as follows:

$$\begin{aligned} & S^* = (MI^{S^*}, 0, \gamma^{S^*}) \\ & S = S^* \cup S \end{aligned} \tag{6.3}$$

Case 1: Add a new service with input from a service and output to sink. This change happens when there is a need to process the output of parent service further, provide new analytical insight or perform verification on the result by repeating the processing. For instance, in Figure 6.1, traffic optimisation or weather predication service can be added based on additional processing of input streaming data from traffic management controller service. With this case, the following are user-supplied input and impact:

User Input: A new service S^* as well as the input link from parent service S_n to new service S^*.

Impact: The S^* and e_{m+1} are created, and S and E sets are updated as follows:

$$\begin{aligned} &Eq.6.3 \textit{ is applied} \\ &e_{m+1} = (S_n, S^*, 100\%) \\ &E = e_{m+1} \cup E \end{aligned} \tag{6.4}$$

Case 2: Add a new service with input from two or more sources (external source and/or parent service) and output to sink. This change happens when a new service is needed to process streaming data combined from several sources and produces new analytical insight. For example, in Figure 6.1, traffic predication service that is based on inputs from traffic management controller (dynamic flows) and traffic modelling (static flows) can be added. With this case, the following are user-supplied input and impact:

User Input: A new service S^* as well as $SS(S^*)$ denotes a set of J input sources for S^*, which is subset of sets EX and S, $SS(S^*) \subseteq EX \cup S$. Thus, each source x_j of set $SS(S^*)$ for $j = 1, 2, .., n$ is either external source $x_j \in EX$ or parent service $x_j \in S$.

Impact: The S^* is created, and S and E sets are updated as follows:

$$\begin{aligned} &Eq.6.3 \textit{ is applied} \\ &\forall x_j \in SS(S^*), e_{m+j} = (x_j, S^*, 100\%) \\ &E = e_{m+j} \cup E \end{aligned} \tag{6.5}$$

Case 3: Add a new service with input from external source and output to one service. This change happens when streaming data from external source can be processed with different logic to provide additional analytics that improve the data analysis performed by an existing service. It also could happen when data preprocessing analytical component/service for streaming data from an existing external source is required prior carrying out data analysis processing on such data by an existing service. For example, in Figure 6.1, a road weather data preprocessing service is needed to perform major data preprocessing steps for road weather sensor data before road weather analysis service in order to significantly reduce data processing time and cost. With this case, the following are user-supplied input and impact:

User Input:

A new service S^* as well as one of the existing services is selected as a destination service

S_n such that this service has input from external source and output to one service. Let EX_p is input source for S_n where $e_m = (EX_p, S_n, 100\%)$ is exist.

Impact: The S^* and two new edges $e_{m+1}\&e_{m+2}$ are created, e_m is deleted, and S and E sets are updated as follows:

$$\begin{aligned}
&Eq.6.3 \textit{ is applied}\\
&E = E - e_m\\
&e_{m+1} = (EX_p, S^*, 100\%)\\
&e_{m+2} = (S^*, S_n, 100\%)\\
&E = e_{m+1} \cup e_{m+2} \cup E\\
&\forall x \in S \textit{ affected by this change, } inStream(x),\\
&outStream(x) \textit{ and } pro(x) \textit{ are updated according to}\\
&Eq.5.6 \textit{ and } Eq.5.7.
\end{aligned} \tag{6.6}$$

Case 4: Add a new service with input from service and output to one service. This change happens when input streaming data of an existing service should be preprocessed before carrying out data analysis processing at this service, so that a new data preprocessing analytical component/service before the existing service is needed. For example, in Figure 6.1, a new service to transform streaming data from vehicle detection service is needed prior to being classified and aggregated by vehicle classification & aggregation service. With this case, the following are user-supplied input and impact:

User Input: A new service S^* as well as one of the existing services is selected as a destination service S_n such that this service has input from one service and output to one or more services. Let S_{n-1} is input source (parent service) for S_n where $e_m = (S_{n-1}, S_n, 100\%)$ is exist.

Impact: The S^* and two new edges $e_{m+1}\&e_{m+2}$ are created, e_m is deleted, and S and E sets are updated as follows:

$$\begin{aligned}
&Eq.6.3 \textit{ is applied}\\
&E = E - e_m\\
&e_{m+1} = (S_{n-1}, S^*, 100\%)\\
&e_{m+2} = (S^*, S_n, 100\%)\\
&E = e_{m+1} \cup e_{m+2} \cup E\\
&\forall x \in S \textit{ that affected by this change, } inStream(x),\\
&outStream(x) \textit{ and } pro(x) \textit{ are updated according to}\\
&Eq.5.6 \textit{ and } Eq.5.7.
\end{aligned} \tag{6.7}$$

Case 5: Add a new service with input from two or more sources and output to one service. This change happens when input streaming data of an existing service needs to be preprocessed and then enriched with additional stream output sources, so that data analysis processing at this service will be carried-out on the analytical result instead

of original streams. To apply this change, a new data analytical component/service before the existing service is necessary. For example, in Figure 6.1, input data of vehicle velocity classification and aggregation service from vehicle detection service is preprocessed and enriched with roadside cameras by using a new service. This new service performs filtering and error correction on such data, where the output stream of this service will then be injected into velocity classification and aggregation service instead of original stream from vehicle detection service. With this case, the following are user-supplied input and impact:

User Input:

A new service S^*, one of existing services is selected as a destination service such that this service has input from one service and output to one or more services, and $SS(S^*)$ denotes a set of J input sources for S^*, which is subset of sets EX and S excluding S_n and S_{n-1}, $SS(S^*) \subseteq EX \cup S - \{S_n, S_{n-1}\}$. Thus, each source x_j of set $SS(S^*)$ for $j = 1, 2, .., n$ is either external source $x_j \in EX$ or parent service $x_j \in S$. Let S_{n-1} is input source (parent service) for S_n where $e_m = (S_{n-1}, S_n, 100\%)$ is exist.

Impact: The S^* and new edges $e_{m+1...(m+|SS(S^*)|)}$ are created, e_m is deleted, and S and E sets are updated as follows:

$$
\begin{aligned}
&\textit{Eq.6.3 is applied}\\
&E = E - e_m\\
&SS(S^*) = SS(S^*) \cup S_{n-1}\\
&\forall x_j \in SS(S^*), e_{m+j} = (x_j, S^*, 100\%)\\
&\qquad E = e_{m+j} \cup E\\
&e_{m+1} = (S^*, S_n, 100\%)\\
&\forall x \in S \textit{ that affected by this change, } inStream(x),\\
&outStream(x) \textit{ and } pro(x) \textit{ are updated according to}\\
&\textit{Eq.5.6 and Eq.5.7.}
\end{aligned}
\tag{6.8}
$$

6.3.4 Dynamic Form 4: Delete an Existing Service

Like adding a new service to workflow application, deleting an existing service also changes the structure of this application. However, the deletion of service varies in cases depending on the output link(s) of this service, where the impact happens on sub-tree of this service in case of output link(s) is not to sink. Under this dynamic form, we support two cases for performing application structure change by the mean of deleting an existing service.

Case 1: Delete a service with output to sink. This case happens in real world when one of ending services in stream workflow application as an analytical component is no longer needed in data analysis pipeline. For instance, traffic accident risks predication service in Figure 6.1 is no longer wanted. With this case, the following are user-supplied input and impact:

User Input: Existing service S_n selected. Let $SS(S_n)$ denote a set of J input sources for S_n, which is subset of sets EX and S excluding S_n, $SS(S_n) \subseteq EX \cup S - S_n$.

Impact: The S_n is deleted, and S and E sets are updated as follows:

$$\begin{aligned} &\forall x_j \in SS(S_n), E = E - (x_j, S_n, 100\%) \\ &pro(S_n) = \emptyset \\ &S = S - S_n \end{aligned} \tag{6.9}$$

Case 2: Delete service with output link(s) to one or more services – This change happens when the analytical processing carried-out by existing service and all subsequent analysis performed by services in the sub-tree of this service are not needed. For example, traffic modelling service and its sub-tree in Figure 6.1 may not be required as analytical components in data pipeline. With this case, the following user-supplied input and impact:
User Input: Existing service S_n selected. Let $SS(S_n)$ denote a set of J input sources for S_n, which is subset of sets EX and S excluding S_n, $SS(S_n) \subseteq EX \cup S - S_n$, and $ST(S_n)$ denote a set of services in the sub-tree of S_n, where each child service S_j of set $ST(S_n)$ for $j = 1, 2, .., n$ is descendant of the current tree service S_n.
Impact: The S_n is deleted, and S and E sets are updated as follows:

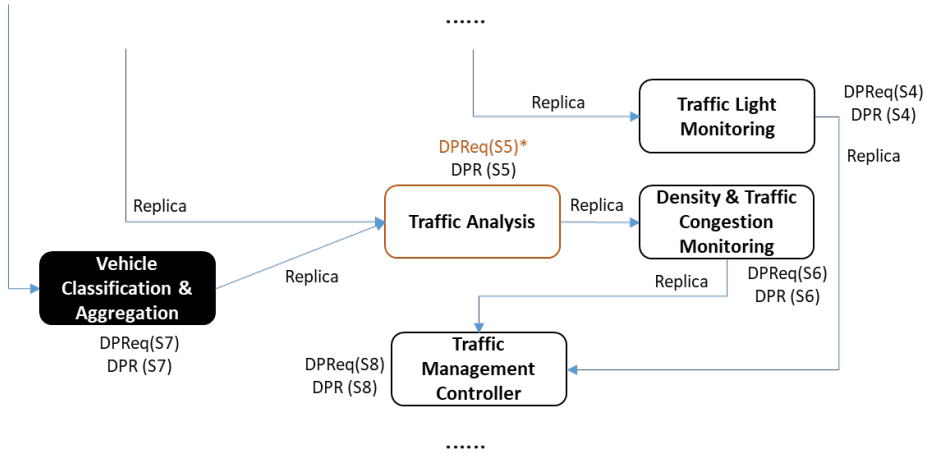
$$\begin{aligned} &\forall x_j \in SS(S_n), E = E - (x_j, S_n, 100\%) \\ &pro(S_n) = \emptyset \\ &\forall S_j \in ST(S_n), S = S - S_j \end{aligned} \tag{6.10}$$

Figure 6.2 to Figure 6.4 show the illustrative case examples of dynamic form 2, 3 and 4 respectively.

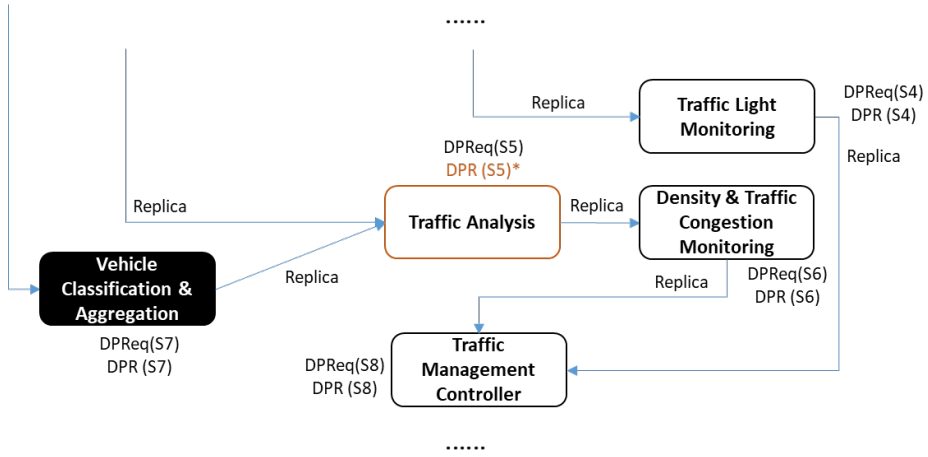
6.4 Proposed Pluggable Scheduling Technique

For adaptive workflows like stream workflows, finding optimal or near-optimal solution at deployment time is not the whole story. This because the dynamic nature of these workflows cause different types of changes to occur at runtime. Tackling runtime changes of in-progress stream workflows needs to be investigated. In this chapter, our problem is to reschedule stream workflow application in cloud infrastructures to respond to different dynamic forms that occur at runtime. The revised scheduling plan should be generated as quick as possible, be cost-effective, and maintain performance requirements. In other words, maintaining the quality of the revised scheduling solution is what matters here.

Considering various dynamic forms of stream workflows, each one of them requires different type of response and this response should be efficient. Consequently, we propose a pluggable dynamic scheduling technique that supports runtime changes of in-progress stream workflows. It handles application-level changes during the execution of this workflow to always guarantee user-defined performance requirements while minimising the execution cost. The proposed technique copes with the four dynamic forms with their different cases as described in structural change modelling (see Section 6.3). The pseudocode of proposed



(a) Case 1: Change data processing requirement of Traffic Analysis Service



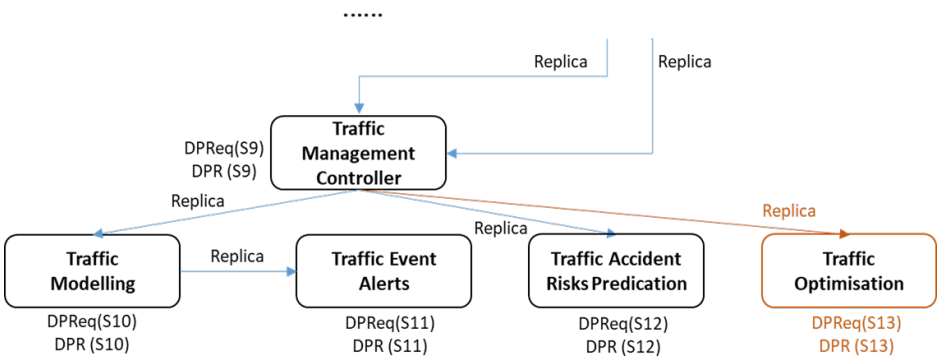
(b) Case 2: Change output stream velocity of Traffic Analysis Service

Figure 6.2: Illustration of Dynamic Form 2 with their Cases after Applying Each Case on Figure 6.1

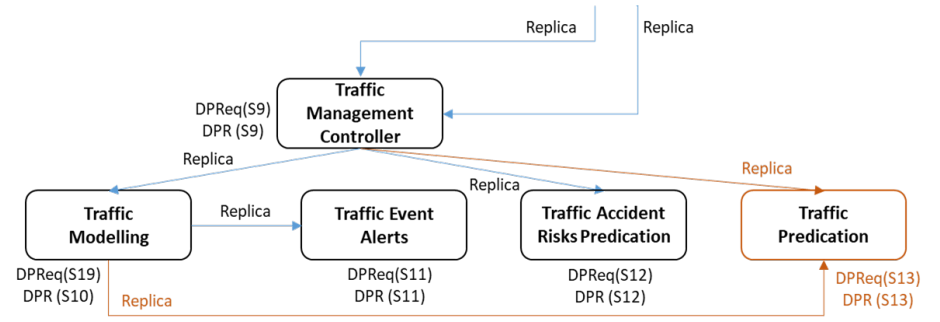
technique is presented in Algorithm 12. This algorithm at the beginning calls the plugged-in scheduling method to generate scheduling plan for deploying the given stream workflow application and then waiting for the occurrence of runtime change. When such change happens, it calls the appropriate method to handle this change.

Algorithm 13 presents the pseudocode of the handling method that is used with dynamic form 2. This algorithm firstly checks whether the change event is increase or decrease change. Then, it retrieves the service to be changed S_n and calculates the change value based on the change percent from original value. After that, it checks the dynamic case, updates the original value and calls the plugged-in algorithm to amend scheduling plan based on dynamic case. It is worth to note that with increase change of data processing requirement for an existing service, the plugin algorithm must evaluate the computing power of the provisioned VM(s) if they are able to maintain the minimum data processing based on the updated data processing requirement for this service. This is because the absence of this check may lead to violating real-time data processing requirement for that service.

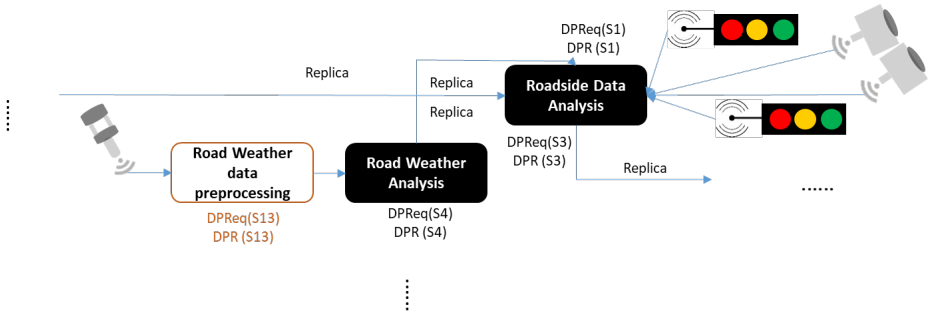
Algorithm 14 presents the pseudocode of the handling method that is used with dynamic



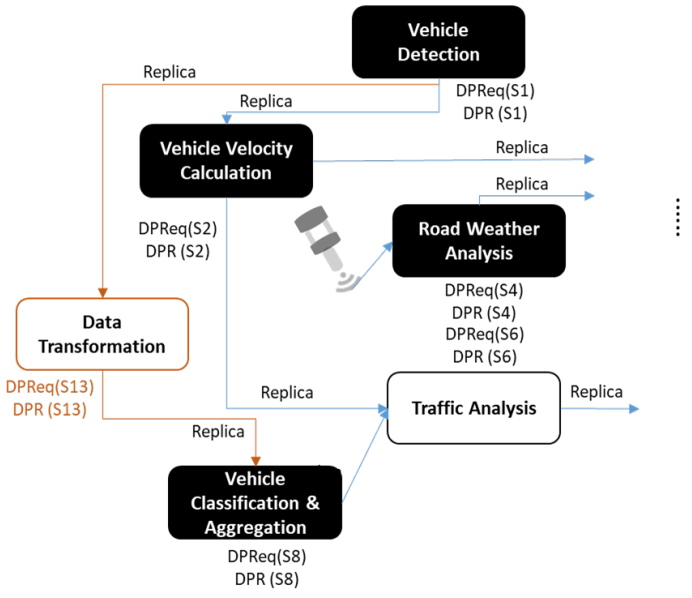
(a) Case 1: Add Traffic Optimisation Service



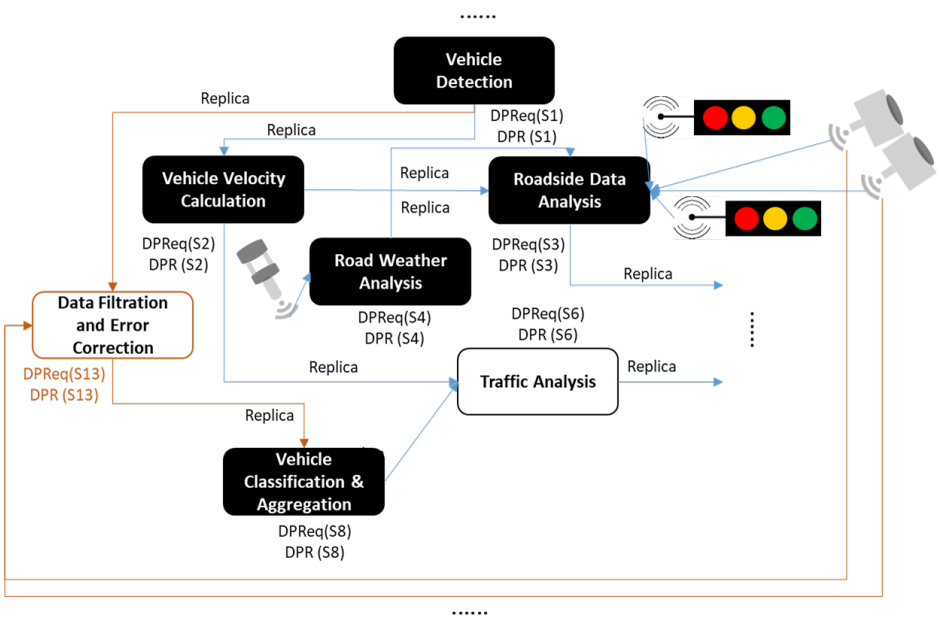
(b) Case 2: Add Traffic Predication Service



(c) Case 3: Add Road Weather Data Preprocessing Service



(d) Case 4: Add Data Transformation Service

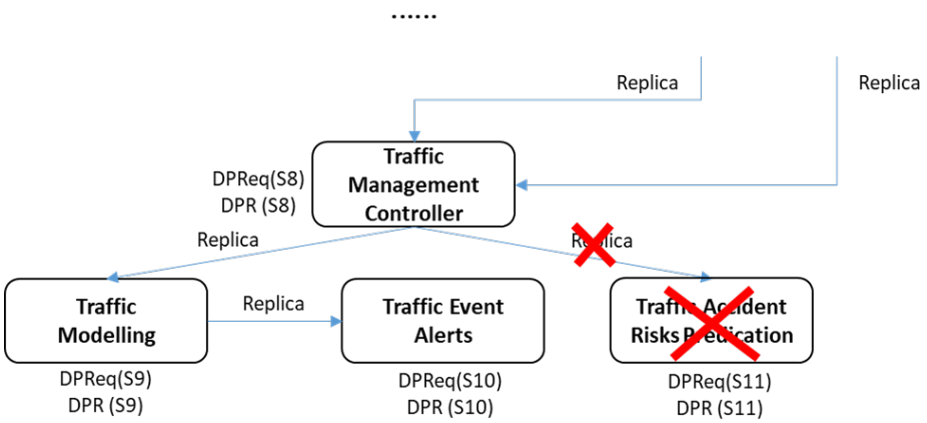


(e) Case 5: Add Data Filtration and Correction Service

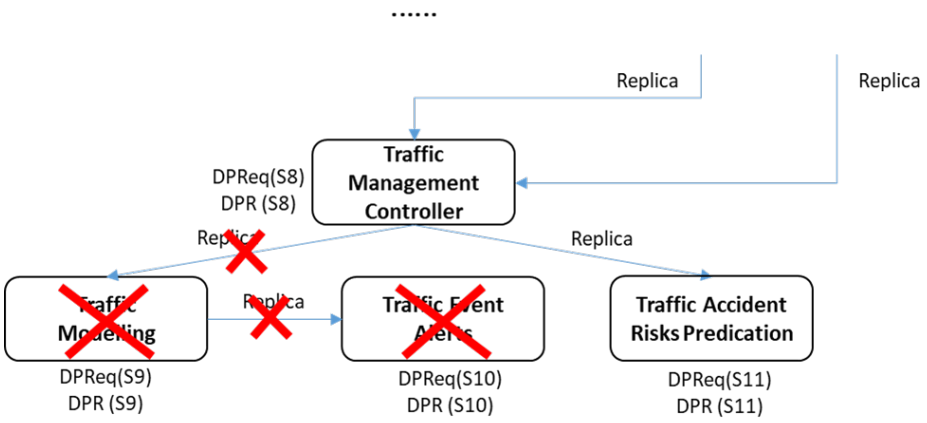
Figure 6.3: Illustration of Dynamic Form 3 with their Cases after Applying Each Case on Figure 6.1

form 3 case 1 and case 2, while Algorithm 15 presents the pseudocode of the handling method for the rest of dynamic form 3 cases (i.e. case 3, case 4 and case 5). Both algorithms create a new service S^* with its type, data processing requirement, placement cloud and output data proportion. But, the main difference between these algorithms is that the former adds sink service while the latter adds non-sink service which has impact on downstream services. Thus, Algorithm 14 retrieves the input data sources selected for the service to be added, adds these input data sources based on the dynamic case to this service and updates parent-child relationships. It then adds output stream and calls handleNewServiceChange method with only one affected service (i.e. the service to be added) in order to deploy it. Algorithm 15 firstly retrieves the destination service S_n, where the new service S^* will be added before that service. Based on the dynamic case, this algorithm retrieves the input data sources selected for the service to be added and adds these input data sources to this service. Then, it updates the stream dependencies and parent-child relationships for input source of destination service S_{n-1}, new service S^* and destination service S_n. After that, it retrieves the list of downstream services that will be affected by the addition of a new service. Lastly, it calls handleNewServiceChange method with this list to amend scheduling plan.

Algorithm 16 present the pseudocode of method that is used to handle all cases of dynamic form 3. This algorithm first retrieves the service to be added S^* and deploys this service using the plugged-in algorithm. Then, it calls the plugged-in algorithm with each service affected from runtime change to amend the provisioning plan of this service. After that, it retrieves those VMs that will be deprovisioned and those VMs that will be provisioned.



(a) Case 1: Delete Traffic Accident Risks Predication Service



(b) Case 2: Delete Traffic Modelling Service

Figure 6.4: Illustration of Dynamic Form 4 with their Cases after Applying Each Case on Figure 6.1

Lastly, it perform provisioning and deprovisioning requests.

Algorithm 17 presents the pseudocode of the handling method that is used with dynamic form 4 case 1. While for dynamic form 4 case 2, the handling method is presented in Algorithm 18. Both algorithms delete an existing service S_n, but the main difference between these algorithms is that the former removes sink service which has impact on the parent service that could become a sink service, while the latter removes non-sink service in which the sub-tree of this service should be removed. Algorithm 17 first retrieves the existing service to be deleted S_n. Then, it removes stream dependencies of this service and updates the child relationships of input data sources of this service. After that, it deprovision VM(s) allocated to that service and lastly deletes the service. Similarly, Algorithm 18 performs the same actions, and in addition to the need of retrieving the sub-tree of S_n, it deletes those services in this sub-tree by deprovisioning their VMs, removing their stream dependencies and parent-child relationships.

Algorithm 12 Pluggable Dynamic Scheduling Technique

call Scheduling Algorithm for deployment of stream workflow {a plugin algorithm that finds resource selection and scheduling solution at deplyment time for executing stream workflow}
for each runtime change that occurs **do**
  $appChangeType \leftarrow$ get dynamic form
  $appChangeCase \leftarrow$ get dynamic case in this dynamic form
  if appChangeType == 1 **then**
    call Velocity Change Response Algorithm {a plugin algorithm to revise the current scheduling plan to cope with data velocity changes}
  else if appChangeType == 2 **then**
    call processExistingServiceChange(appChangeCase);
  else if appChangeType == 3 **then**
    if appChangeCase == 1 or appChangeCase == 2 **then**
      call newServiceChange_Case1&2(appChangeCase);
    else if appChangeCase == 3 or appChangeCase == 4 or appChangeCase == 5 **then**
      call newServiceChange_Case3&4&5(appChangeCase);
    end if
  else if appChangeType == 4 **then**
    if appChangeCase == 1 **then**
      deleteService_Case1();
    else if appChangeCase == 2 **then**
      deleteService_Case2();
    end if
  end if
end for

Algorithm 13 processExistingServiceChange(appChangeCase)

$changeEvent \leftarrow$ get change event type {increase or decrease}
$S_n \leftarrow$ get the existing service that is selected
get provisioned VMs for a service
$changePercent \leftarrow$ get increase/decrease percent form original value
$value \leftarrow changePercent/100$
if appChangeCase == 1 **then**
  if changeEvent == 'increase' **then**
    $MI^{S_n} = MI^{S_n} + (MI^{S_n} * value)$
  else
    $MI^{S_n} = MI^{S_n} * value$ $\{0.01 < value < 0.99\}$
  end if
  call Data Processing Requirement Change Response Algorithm with given service S_n {a plugin algorithm that assesses each provisioned VMs as still satisfying minimum data processing based on the updated MI^{S_n} and provisioning more computing power if needed in case of increase event or deprovisioning the provisioned VMs that are not required to achieve the updated MI^{S_n} in case of decrease event}
end if
if appChangeCase == 2 **then**
  if changeEvent == 'increase' **then**
    $\gamma^{S_n} = \gamma^{S_n} + \gamma^{S_n} * value)$
    call Velocity Change Response Algorithm with increase event {a plugin algorithm to amend scheduling plan to handle data velocity changes}
  else
    $\gamma^{S_n} = \gamma^{S_n} * value$ $\{0.01 < value < 0.99\}$
    call Velocity Change Response Algorithm with decrease event {a plugin algorithm to revise scheduling plan to handle data velocity changes}
  end if
end if

Algorithm 14 newServiceChange_Case1&2(appChangeCase)

Create service S^*
InputSources = ϕ
Add input source(s) selected to S^* in InputSources {Case 1: one input source, Case 2: two or more input sources; selection constraints are applied}
for each input source in InputSources **do**
    Add input dependency to S^* from this source
end for
Update parent and child relationships for input source(s) and S^*
Add output stream for S^* based on input stream dependency
Add S^* to affectedSIDs
call handleNewServiceChange(affectedSIDs)

Algorithm 15 newServiceChange_Case3&4&5(appChangeCase)

Create service S^*
$S_n \leftarrow$ get one of the existing services as a destination service {selection constraint is applied according to dynamic case}
InputSources = ϕ
if appChangeCase == 3 or appChangeCase == 4 **then**
    Add input source of S_n (i.e. S_{n-1}) in InputSources
else
    Add input source(s) selected to S^* in InputSources {input source of S_n (i.e. S_{n-1}) + other input sources that are selected}
end if
for each input source in InputSources **do**
    Add input dependency to S^* from this source
end for
Remove the dependency link of S_n
Add output stream for S^* based on input stream dependency
Add dependency link for S_n {source: S^*}
Update parent and child relationships between service(s) in InputSources and S^*, and between S^* and S_n
affectedSIDs = get ids of services affected by adding request starting from S^*
call handleNewServiceChange(affectedSIDs)

Algorithm 16 handleNewServiceChange(affectedSIDs)

$S^* \leftarrow$ get and remove the new service from affectedSIDs
call Resource Selection Algorithm for S^* {a plugin algorithm that finds near-optimal resource selection solution for given service}
if affectedSIDs is not empty **then**
    call Velocity Change Response Algorithm with the list of affected services (affectedSIDs) {a plugin algorithm to revise the current scheduling plan to cope with data velocity changes for the list of services provided}
end if
$BeProVMs \leftarrow$ get VMs that need to be provisioned S^*
$BeDeproVMs \leftarrow$ get VMs that need to be deprovisioned S^*
if BeProVMs is not empty **then**
    provision VMs in the list
else
    if BeDeproVMs is not empty **then**
        deprovision VMs in the list
    end if
end if

Algorithm 17 deleteService_Case1()

1: $S_n \leftarrow$ get existing service that will be deleted {selection constraint is applied}
2: InputSources $\leftarrow$ get input sources of S_n
3: Remove dependency link(s) to S_n from those sources in InputSources
4: Update child relationships for those sources in InputSources
5: Deprovision the provisioned VM(s) of S_n
6: Delete S_n

Algorithm 18 deleteService_Case2()

1: $S_n \leftarrow$ get existing service that will be deleted
2: subtree $\leftarrow$ get services in sub-tree of S_n
3: **for** for each service in subtree **do**
4: Deprovision the provisioned VM(s) of this service
5: Remove this service with its dependency link(s) and parent-child relationships
6: **end for**
7: InputSources $\leftarrow$ get input sources of S_n
8: Remove dependency link(s) to S_n from those sources in InputSources
9: Update child relationships for those sources in InputSources
10: Deprovision the provisioned VM(s) of S_n
11: Delete S_n

6.5 Experiment Setup and Configuration

The aim of our experiments is to assess the quality of the response that is generated when a dynamic change happens at runtime. For this purpose, we simulate Multicloud environment with the proposed pluggable dynamic scheduling technique using the proposed IoTSimStream. The simulation environment used facilitates our evaluation as we can compare experimental results obtained from different scheduling algorithms under the same environment conditions.

6.5.1 Workflow Application and Simulation Environment

In Section 4.5.1, we provided stream workflow applications that modelled using common workflow structures (Montage, Inspiral, Epigenomics and CyberShake) and Multicloud environment that is made up of three different clouds (Amazon EC2, Google Cloud Engine and Microsoft Azure), where each cloud offers different VM configurations. These applications with their different configuration sizes and this modelled execution environment are used in our experiments. Additionally, a set of parameters for both workflow application and simulator should be configured to run our experiments. Thus, we used workflow and simulation parameters and their values presented in Table 5.6 for all our scenarios. For "External Source Data Rate" parameter, we consider here the data rate range [5, 10] MB/s to obtain the value.

6.5.2 Configuration Changes in Service Data Processing Requirement

To model the amount of increase or decrease in service data processing requirement, we do not only consider the percent increase of the original value up to 100%, but we add 50% as an additional margin to make it 150%. With the highest increase percent (i.e. upto 150%), data processing requirement for simple or medium aggregation function will be transformed to represent complex aggregation function. Thus, it is valuable to take into consideration the additional margin to increase data processing requirement. We also need to note that the updated value of data processing requirement after increase request is capped at 4000 MI/MB, which is the maximum value of data processing requirement as listed in Chapter 4. For decreasing data processing requirement, we model the percent decrease of upto 75%, where 25% is considered as additional margin to transform data processing requirement of complex aggregation function into simple aggregate function. Table 6.1 and Table 6.2 list the percentages of change to increase and decrease data processing requirement respectively.

Table 6.1: Percentage ranges of service data processing requirement increase

Range	**Minimum (Percent)**	**Maximum (Percent)**
Low	10	40
Medium	60	90
High	110	150

Table 6.2: Percentage ranges of service data processing requirement decrease

Range	**Minimum (Percent)**	**Maximum (Percent)**
Low	5	25
Medium	35	55
High	65	75

6.5.3 Configuration Changes in Service Output Data Rate

Considering S. Rizou et al. research work (Rizou et al., 2010), the selectivity of the operator is the percent of output data to input data and this selectivity is varying between 0 to 1. The selectivity close to 1 means that the operator generates output data rate equals to the input data rate, while selectivity close to 0 means that the operator generates very low output data rate and acts as high selective filter in the network. In this research work, the output data rate is upto 100% of input data rate and the data rate unit is kilobit per second. The advancements of networking and IoT technologies lead to increase speed of data being exchanged. For example, in L. Fischer and A. Bernstein (Fischer and Bernstein, 2015), the size of tuple used in experiments is varying from bytes to KB to MB so that the data rate unit in megabit / megabyte per second is being considered. These advancements are continuous which means the data rate will keep increase. Therefore, it is valuable to consider additional 50% beyond 100% as the future margin of increase. Accordingly, the output data rate is considered to be upto 150% of input data rate as defined in Chapter 4.

Considering output data rate is upto 150% of input data rate (including 50% additional margin), we consider the percent change in increasing service output data rate is upto 100% while for decreasing service output data rate is upto 75%. The percentage ranges of both service output data rate increase and decrease amounts are presented in Table 6.3 and Table 6.4.

Table 6.3: Percentage ranges of service output data rate increase

Range	**Minimum (Percent)**	**Maximum (Percent)**
Low	10	30
Medium	50	70
High	90	100

Table 6.4: Percentage ranges of service output data rate decrease amount

Range	**Minimum (Percent)**	**Maximum (Percent)**
Low	5	15
Medium	25	35
High	45	50

6.5.4 Experimental Scenarios

To evaluate the quality of response solution to revise the scheduling plan at runtime when application-level change happens, we examine different experiment scenarios for different dynamic forms. All these experiments are with regard to cost, changes and time. The cost is the solution cost after the change applied. This cost includes data provisioning cost and data transfer cost. Regarding to the change, we consider the number of changes applied in the current scheduling plan in term of compute resources to respond to any runtime change. In other words, it is a number of VM provisioning and/or deprovisioning changes that are made for revising the current scheduling plan. For the time, we consider the request execution time (computational time) required to process and complete this request. This time is a sum of request's processing time, algorithm running time and highest boot time among the VMs provisioned.

For each dynamic case, we conduct an experiment to study the quality of solutions being generated in respond to this dynamic change. Thus, we will perform 11 experiments ad follows: two experiments for each case in dynamic form 2 (one for increase request and the other for decrease request, with a total of 4 experiments), one experiment for each case in dynamic 3 (with a total of 5 experiments) and one experiment for each case in dynamic 4 (with a total of 2 experiments). Then, we record experimental results of the quality of solution and then comparing these results in order to examine the scale of performance quality.

The aforementioned experiment scenarios are used to examine and evaluate the performance and service quality of three different techniques under different application-level changes.

Baseline Technique (BT): our pluggable dynamic scheduling technique with proposed realistic and straightforward algorithm (baseline algorithm) that does not need to use any complicated heuristic. This algorithm handles different dynamic changes by simply provisioning VM with highest computing power and achieve service minimum data processing unit when a new service is deployed or more computing power is needed, and deprovisioning part of all VMs available when existing service is deleted, or less computing power is needed. It worth to note that this technique can, if possible, deprovision part of VMs available based on the amount of computing power being decreased.

Dynamic Fair-Share Technique (DFST): Fair sharing model (to one resource type or multiple resource types) is a default scheduling decision used by Apache YARN and Mesos to equal share the resources of cluster among applications over time. This default scheduler cannot handle dynamic forms of stream workflows by managing the resources at runtime. Therefore, we have extended and implemented this model to support elasticity and adjust scheduling plan at runtime to cope with stream workflows and its dynamic forms. Accordingly, DFST is our pluggable dynamic scheduling technique with the proposed dynamic fairness heuristic method. This technique provisions the same type of VM (with high-medium computing power) when a new service is deployed or there is a need for more computing power, and deprovisions any available VM when less computing power is needed or releases all of available VM(s) when the service being deleted.

Optimisation Technique (OT): our pluggable dynamic scheduling technique with proposed plugin algorithms and methods that presented in Section 6.5.5.

By comparing the quality of solution being generated by our techniques (BT, DFST and OT) in respond to various dynamic changes, we can evaluate the efficiency of each technique in respect to others and find the most efficient technique that produced the best response solution. The comparison between OT and BT is aimed at figuring-out whether the complex heuristic-based method is necessary to improve the quality of solution being generated to respond to application-level runtime change. While the comparison between OT and DFST is aimed at evaluating the quality of solution generated by the developed dynamic version of fair-share scheduling decision since fair share model is used in big data application orchestrators (i.e. Apache YARN and Mesos).

Note that we will not conduct experiment for dynamic form 1 (change the streaming data velocity) as we already investigated this form in detail in Chapter 5.

6.5.5 Plugin Scheduling Algorithms and Techniques

As the proposed technique is a pluggable method to dynamically schedule stream workflow at runtime, customisable or plugin scheduling algorithms are needed to make scheduling decisions. To run our experiments, different types of plugin algorithms/methods are used with different techniques to perform quality of solution evaluations according to the aforementioned experimental scenarios.

With OT, we use the proposed scheduling algorithm and techniques in Chapter 4 and 5.

For scheduling stream workflow at deployment time, the proposed algorithms in Chapter 4 or GA with random immigrants scheme in Chapter 4 can be used as plugin algorithm. GA with Random Immigrants Scheme is the advanced version of traditional GA and performed better even with dynamic scheduling according to the results presented in Chapter 5, we use it as plugin algorithm in Line 1 of Algorithm 12.

For adaptive scheduling with dynamic form 1 and 2, the proposed two-level greedy algorithm in Chapter 5 is used as a plugin algorithm. For dynamic form 1, this algorithm is used in Line 6 of Algorithm 12 to dynamically respond to the data velocity changes for services by finding the best resource selection solution for those services affected by such changes. For dynamic form 2, it is used in Line 17 and 20 of Algorithm 13 to revise scheduling plan with both data velocity events. With velocity increase event, this algorithm is called for provisioning more computing powers while with velocity decrease event, it finds those VMs that are not needed any more to deprovision them. In Line 12 of Algorithm 13, a new heuristic technique is proposed as plugin algorithm (see Algorithm 19). It assesses all provisioned VMs of a given service to ensure they still achieves minimum data processing based on the updated data processing requirement. Then it provisions more computing power if needed in case of increase event or deprovisioning those provisioned VMs that are not needed to achieve the updated data processing requirement in case of decrease event.

For adaptive scheduling with dynamic form 3, two algorithms are used as plugin algorithms. The proposed greedy selection algorithm in Chapter 4 is used in Line 2 of Algorithm 16 to find computing resources (resource selection solution) for the new service. Note that, this algorithm originally works on all services, but for the purpose here, we made a minor modification to make it run only with given service (i.e. new service). Moreover, in Line 4 of Algorithm 16, the proposed two-level greedy algorithm in Chapter 5 is used as a pluggable algorithm to revise scheduling plan based on the list of services affected by dynamic change.

With BT, the proposed technique is plugged-in with simple schedule model to provision the highest VM when more computing power is needed and deprovision VM(s) if applicable when the provisioned VM(s) is unnecessary. While with DFST, the proposed technique is plugged-in with the extended fair-share model to support dynamic scheduling. If more computing power is needed (such as adding new service or modifying the existing service with increase request), it provisions the same type of VM, while deprovision VM(s) if applicable when the provisioned VM(s) is unnecessary. For both BT and DFST, if the changes occur in the existing service, they provision and/or deprovision VM(s) according to whether the change is increase request or decrease request after checking the current computing power for this service.

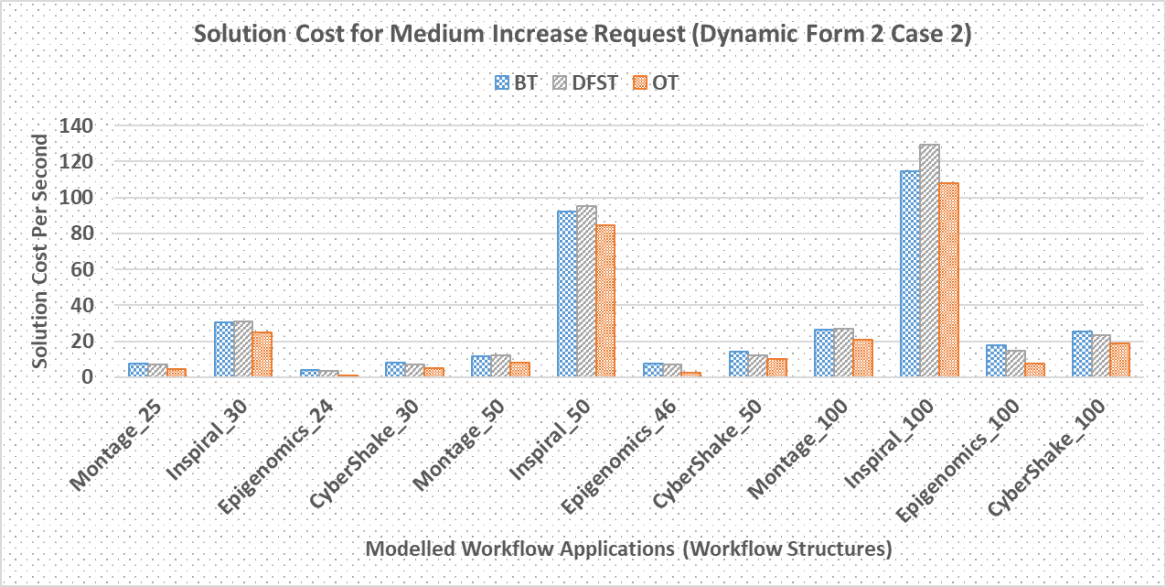
Algorithm 19 ResourceSelection_DPReqChange(Service)

1: $unitMIPS \leftarrow MI^{S_n} * unitDPRate$
2: $reqUnits \leftarrow$ get number of units required based on updated MI^{S_n}
3: $changeEvent \leftarrow$ *get velocity change event* {increase or decrease}
4: $S_n \leftarrow$ *get the existing service that is selected*
5: $pro(S_n) \leftarrow$ provisioned VMs for a service
6: **for** each vm in $pro(S_n)$ **do**
7: **if** $MIPS_{vm} >= unitMIPS$ **then**
8: **if** $reqUnits > 0$ **then**
9: $reqUnits = reqUnits - \lfloor (MIPS_{vm}/unitMIPS) \rfloor$
10: **else**
11: *add vm in* $rmVM(S_n)$ {deprovision vm}
12: **end if**
13: **else**
14: *add vm in* $rmVM(S_n)$
15: **end if**
16: **end for**
17: **while** $reqUnits > 0$ **do**
18: $selectedVM, VMList = \phi$
19: $VMOffers \leftarrow$ *VM offeres of* S_n *placement cloud order by comp. power*
20: **for** each vm_offer in VMOffers **do**
21: $achievedUnits = \lfloor (MIPS_{vm\_offer}/unitMIPS) \rfloor$
22: **if** $achievedUnits >= reqUnits$ *or vm_offer is last offer* **then**
23: $selectedVM = vm\_offer$
24: *break*
25: **end if**
26: **end for**
27: $VMList = VMList \cup selectedVM$
28: $reqUnits = reqUnits - \lfloor (MIPS_{selectedVM}/unitMIPS) \rfloor$
29: **end while**

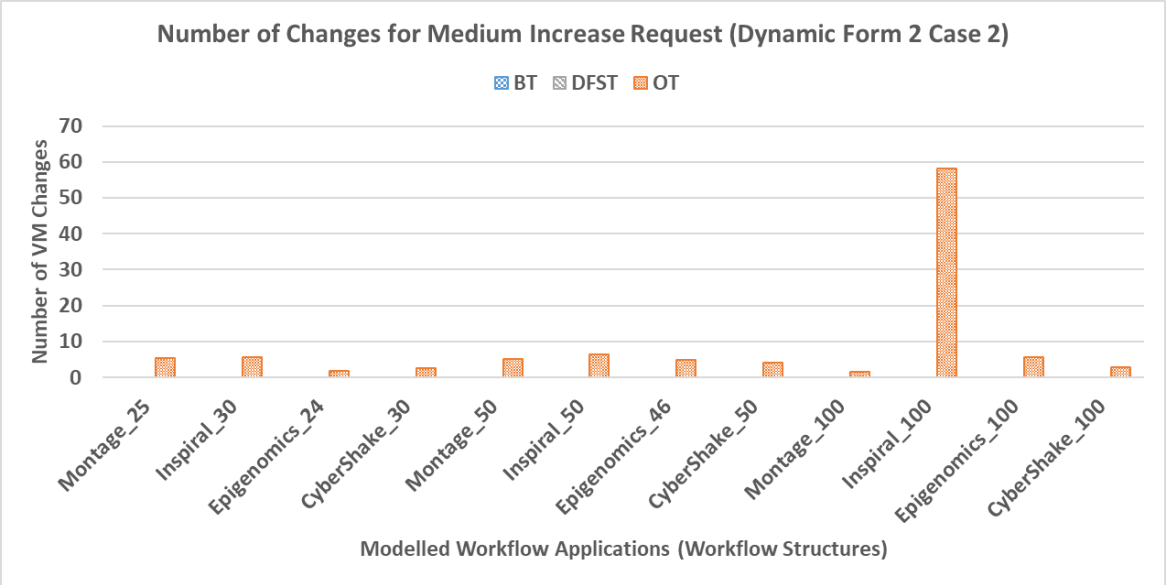
6.6 Experimental Results

To perform our experiments, the proposed IoTSim-Stream is used on a Nectar Cloud virtual machine that had 8 vCPUs, 32GB of RAM memory and running Ubuntu 16.04.1 LTS, and the results of these experiments are collected. For OT, each experimental scenario runs ten times since random-based immigrants genetic algorithm is used in this technique, and then the average value of the obtained results is taken and used in the representation of experimental results. Also, for quality of solution results, we present the average value for both solution cost and number of changes as two runtime changes are made during the simulation time. Regarding to the results of dynamic form 2 scenarios, we only present medium-range results as these results are sufficient to get the conclusion.

We have examined the experimental results looking for key results that allow us to reach to the conclusion. We found that the results of experiment scenarios for dynamic form 2 case 2 (with increase change), dynamic form 2 case 2 (with decrease change), dynamic form 3 case 1, dynamic form 3 case 4 and dynamic form 4 case 1 are enough for our discussion. The rest of results are presented in Appendix D. Figure 6.5 to Figure 6.9 depict the quality of solution result for the aforementioned dynamic forms and cases. From these results, our analysis and findings are:



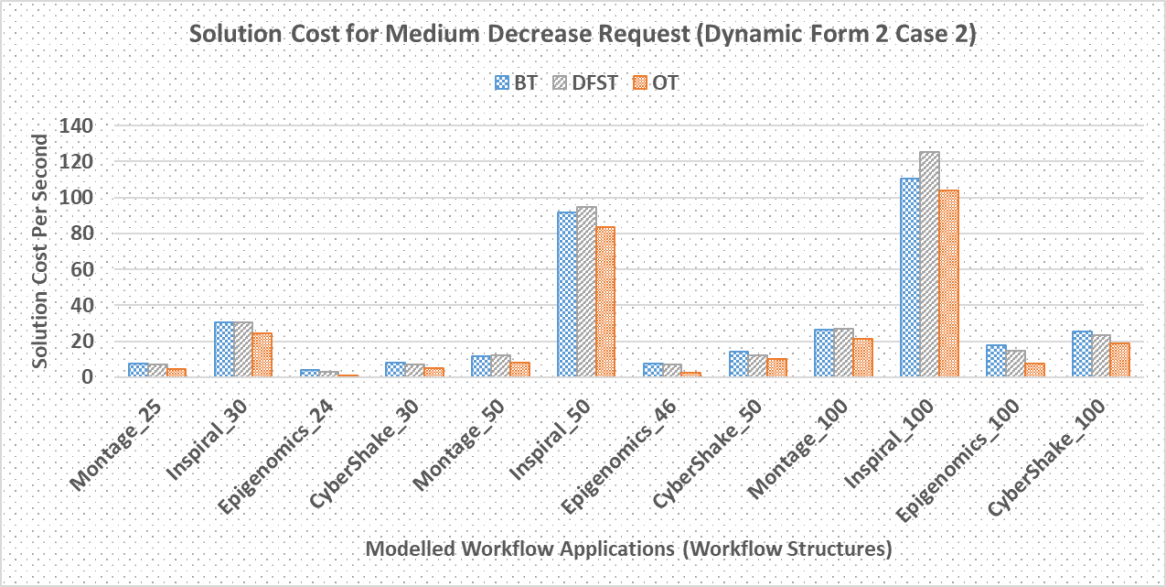
(a) Solution cost



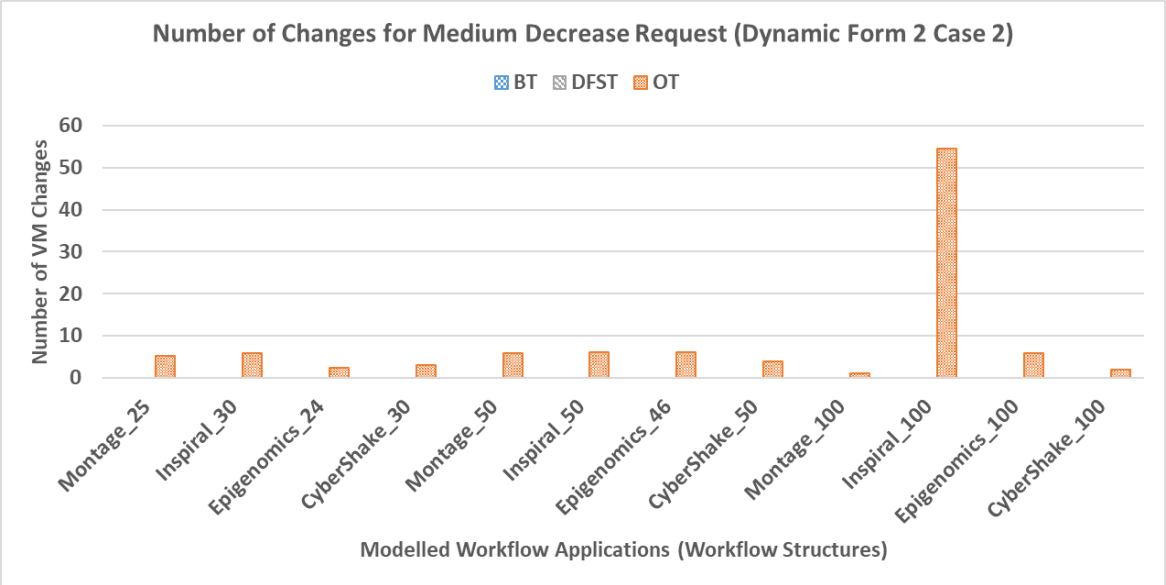
(b) Number of changes

Figure 6.5: Quality of solution for different workflow structures under Dynamic Form 2 Case 2 Increase (medium range)

- In term of solution cost, OT achieved best results in comparison with BT ad DFST. This is because, OT applied genetic algorithm at deployment time to find the best way to place services over multiple cloud infrastructure to minimise not only provisioning cost but also data transfer by utilising data locality. With efficient service placement and scheduling plan, OT is able to maintain the lowest cost when handling all dynamic changes during the execution of workflow. On the other hand, BT and DFST do not have that capability, so they cannot minimise the solution cost after handling dynamic changes. In addition, the most cost savings is achieved by Epigenomics workflow structure. This is because this workflow processes less amount of data compared with



(a) Solution cost

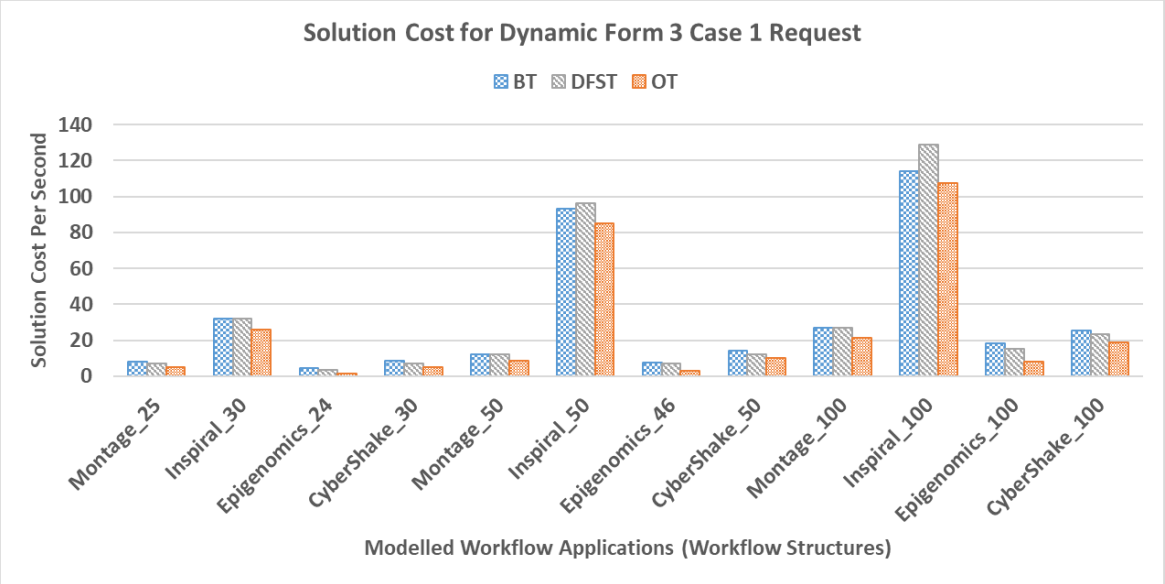


(b) Number of changes

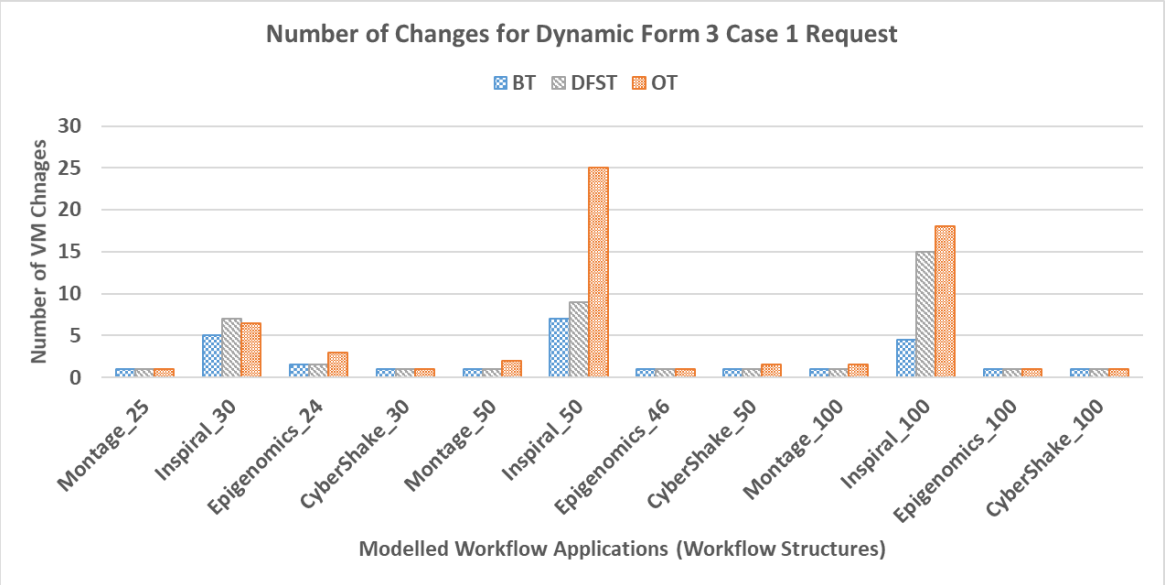
Figure 6.6: Quality of solution for different workflow structures under Dynamic Form 2 Case 2 Decrease (medium range)

other workflow structures, so that less to average computing power is needed. Based on this, provisioning VM with high-medium or highest computing power does not lead to achieve best execution cost, instead finding near-optimal scheduling plan at deployment allow to maximise savings in this step and later when amending the scheduling plan.

- Based on scheduling decision of BT and DFST, the conclusion from Figure 6.5b is that these techniques did not made any VM changes as VM over-provisioning is sufficient to cope with the increase need for computing power in the changed service and downstream services. While, OT needs to make some VM changes as it needs to provision VM(s) with suitable computing power to handle the change request efficiently while



(a) Solution cost

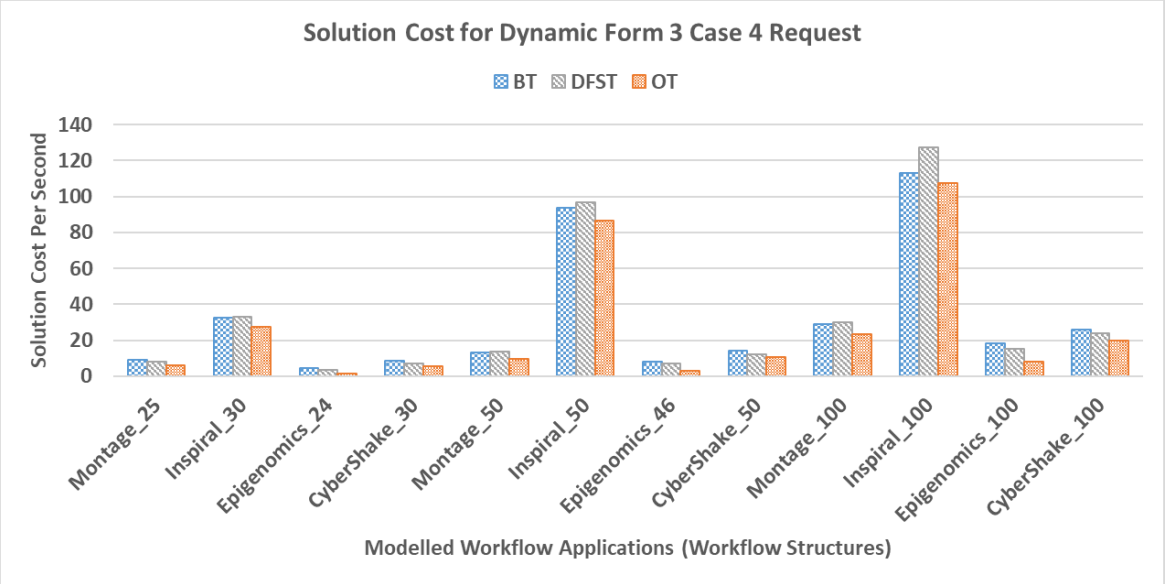


(b) Number of changes

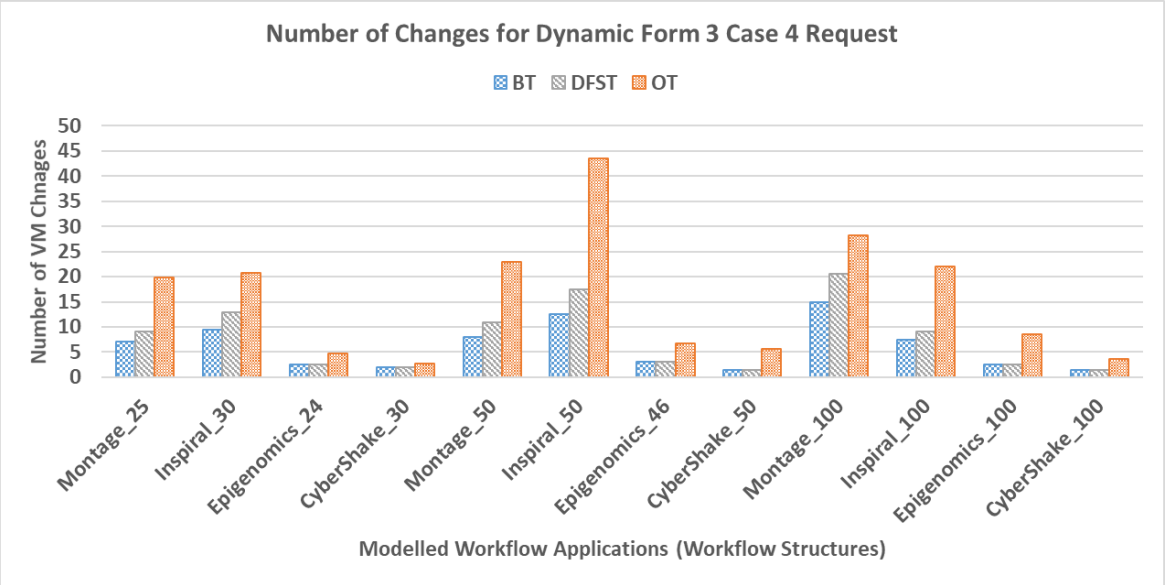
Figure 6.7: Quality of solution for different workflow structures under Dynamic Form 3 Case 1

maintain minimal execution cost.

- From Figure 6.6b, both BT and DFST are unable to deprovision VM(s) with decrease change request. This is because the scheduling decision of these techniques is based on provisioning VM with high-medium to high computing power, so that when there is no substantial decrease in data speed or MIPS value, they cannot avoid over-provisioning and thus additional computing resources are wasted. Moreover, it is worth to note that by having various VM types in the generated scheduling plan like what an OT does, the opportunity becomes very high to find suitable VM(s) to deprovision with any change happens in output stream velocity, leading to avoid over-provisioning and



(a) Solution cost

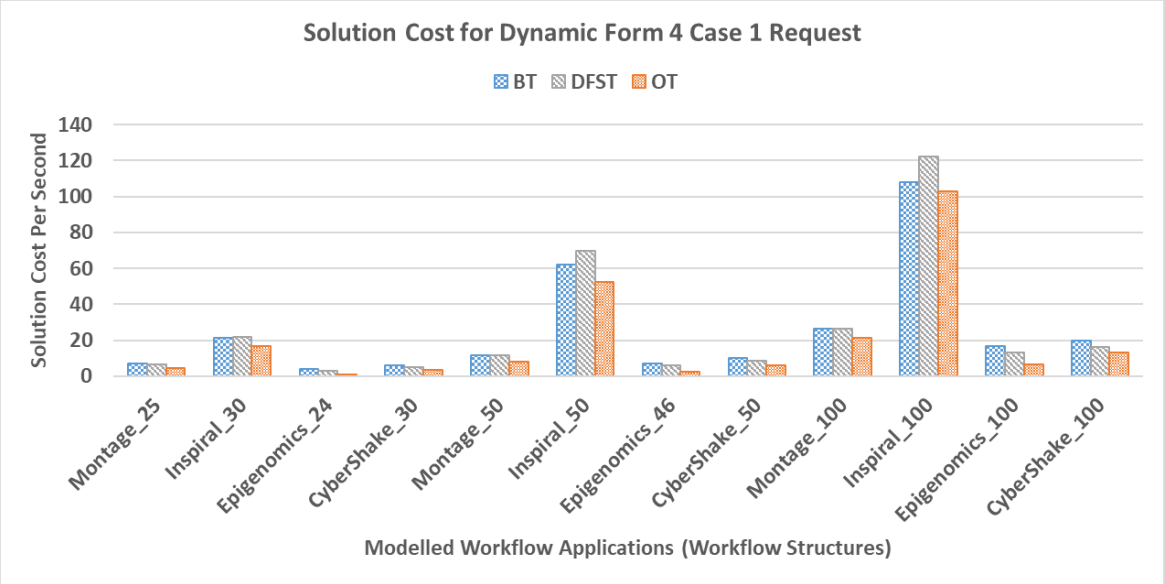


(b) Number of changes

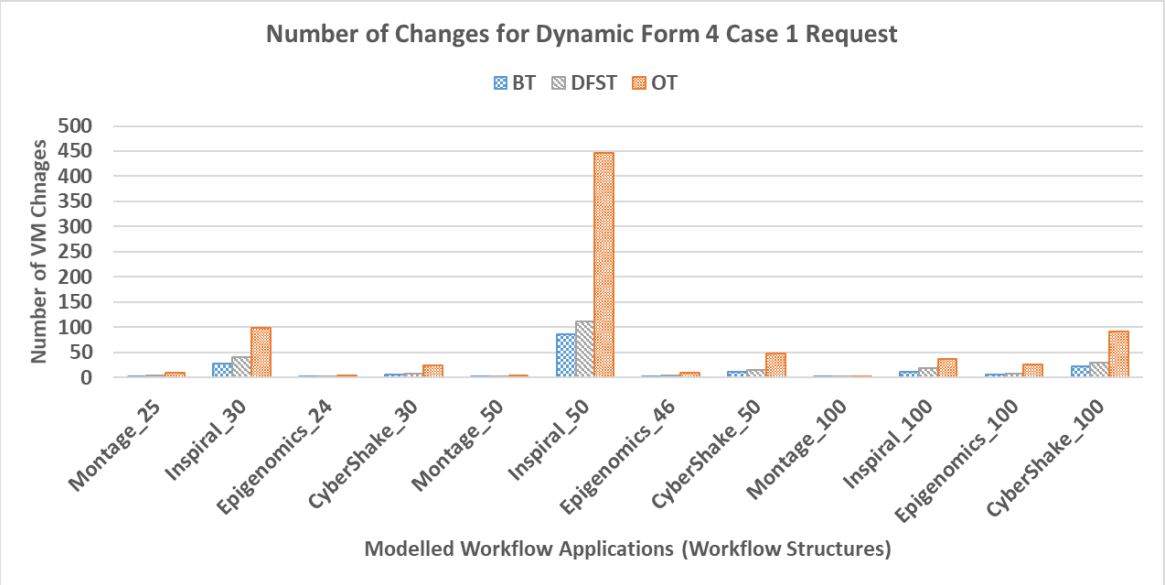
Figure 6.8: Quality of solution for different workflow structures under Dynamic Form 3 Case 4

reduce the total execution cost.

- From Figure 6.7b, it is clear that the results of OT are close to BT and DFST in most cases. This is because adding a new service that outputs to sink has no effect on the other services and it only requires provisioning VMs to deploy this service, so that OT is able to provision suitable VMs while maintaining the minimal number of changes. On the other hand, when the runtime change has an effect on the other services (i.e. downstream services), OT needs to make more VM changes in comparison with BT and DFST (see Figure 6.8b) in order to maintain lower execution costs. Moreover, OT in some cases (such as with Epigenomics_24, CyberShake_30 and CyberShake_100)



(a) Solution cost



(b) Number of changes

Figure 6.9: Quality of solution for different workflow structures under Dynamic Form 4 Case 1

achieved similar results to BT and DFST, as the marginal over-provisioning is sufficient to handle the consequence of adding a new service on the downstream services.

- From Figure 6.9b, the straightforward conclusion is that OT deprovisions more VMs when the service is deleted as it generally provisioned more VMs at deployment time to keeping the execution cost as low as possible by finding a near-optimal/optimal scheduling plan. Moreover, we can notice that the highest number of changes is achieved with Inspiral_50 as this workflow processes a large number of data streams and then requires large computation powers. As such OT at deployment time provisioned more VMs to achieve the required computing powers and deprovisioned them when the service has

deleted.

- In terms of request execution time, which includes computational time required by the scheduling technique to amend the scheduling plan, the VM boot time when more computing power is needed and the time for performing the updates such as updating services, stream dependencies and parent-child relationships. For computational time, all techniques achieved negligible time to amend the scheduling plan since their scheduling decisions are made using a heuristic approach. Also, the time taken for updating is also negligible. Based on this, the request execution time will remain negligible when the technique only needs to deprovision VMs to respond to runtime change (such as dynamic form 4 case 1). While, the request execution time will be increased when there is a need to provision new VMs. This time is mainly determined by the maximum VM boot time among the provisioned VMs. DFST incurred constant time (approx. 35 simulation time units) since this technique provisions the same type of VM all the time. BT incurred on average 36 simulation time units, while OT incurred on average 40 simulation time units. Accordingly, there is no significant difference in request execution time between those techniques.

6.7 Summary

In this chapter, we investigated the problem of dynamically scheduling stream workflow in the cloud under its different dynamic forms including fluctuations of input data rate, changes to workflow structure and changes to real-time data processing requirements. To enable full dynamic support for this workflow, we proposed a scalable and pluggable dynamic scheduling technique that allows the user to plug her/his algorithms and methods in place to respond to runtime changes with the focus on scheduling decision rather than the complexity of dealing with these changes. We also presented three different plug-in techniques that can be used to handle the aforementioned runtime changes, so the user can use these built-in techniques. These techniques vary based on their complexity, from simple heuristic to multi-heuristic algorithm, to tackle different dynamic forms of stream workflow. The quality of solution generated by these techniques are evaluated to determine the most efficient technique. The experimental results showed that the optimisation technique outperformed the baseline technique and the dynamic fair-share technique in quality of solution evaluations.

Chapter 7

Conclusions and Future Works

7.1 Conclusion

This thesis has investigated the optimisation problem of scheduling stream workflow applications and proposed new scheduling techniques to effectively manage the schedules of these applications over cloud infrastructures and handle application-level changes and data fluctuations while meeting user real-time data analysis requirements and minimising the overall execution cost. It began with a literature survey and taxonomy of big data workflow orchestration in the cloud which was presented in Chapter 2. This review highlighted the key requirements and challenges of executing big data workflows in the cloud and revealed a number of research gaps in the study area. One of these gaps is distributed execution of stream workflow applications in cloud environment while meeting real-time user performance requirements and minimising execution cost.

In Chapter 3, we designed and implemented a new IoT simulation toolkit (IoTSim-Stream) that models complex stream workflow applications in cloud computing. This simulator was evaluated and its efficiency is assessed through extensive performance and scalability evaluations. Experimental results showed that performance gains are attainable by calculating and collecting execution performance (in term of execution time, CPU and memory usage) in a large-scale test environment. With IoTSim-Stream, we can conduct large-scale simulation-based studies to evaluate and analyse stream workflow applications with full control over workflow application, cloud computing environment and simulation environment configurations. Therefore, IoTSim-Stream is capable of dealing with the complexities of simulating the execution of stream workflow applications and the simulated environments, even with the most complex configurations.

Following the development and implementation of IoTSim-Stream, the problem of statically scheduling stream workflows in a Multicloud environment was investigated in Chapter 4. This problem was formulated by presenting application and system models. Based on our modelling, we proposed two heuristic algorithms to meet user performance requirements in terms of maximum throughputs while minimising overall execution cost. The proposed

algorithms made scheduling decisions at deployment time to execute given stream workflow application over multiple cloud infrastructures. The efficiency of the proposed algorithms were evaluated in terms of execution costs and their behaviours were studied in terms of computational time and end-to-end latency. With stream workflow applications, end-to-end latency is crucial as it is a major concern for the user. From experimental results, we found that the proposed algorithms kept average end-to-end latency to sub-second times because every data stream that arrives is processed as quick as its data dependences are processed. Also, experimental results showed that the proposed genetic algorithm achieved the best execution cost and relatively low computational time across all scenarios. This indicates the proposed genetic algorithm is efficient in searching and exploring complex search spaces, and then finding optimal/near-optimal provisioning and scheduling solutions under seven varying parameters that were studied in this chapter. Furthermore, experimental results showed that applying high placement restrictions for services narrowed the opportunity for cost reduction. In this case, the proposed genetic algorithm may not be able to find the best solution. Overall, the proposed genetic algorithm is more efficient than the proposed greedy as it manages the trade-off between execution cost, computational time and end-to-end latency.

In Chapter 5 and 6, this thesis studied the problem of dynamically scheduling stream workflows in a Multicloud environment. Chapter 5 discussed the scheduling problem in a scenario, where data velocity changes over time, so that the load of incoming data streams at a given time determines the required computing resources. This dynamic scheduling problem is modelled, and based on this model, an adaptive scheduling technique was proposed to revise scheduling plan at runtime while achieving real-time performance requirements under varying data rates in a cost-effective manner. Extensive experimental studies were conducted to assess the efficiency and performance of the proposed technique. There were three types of experiment conducted. The first set of experiments was to compare the proposed technique with a baseline, genetic algorithm and lower bound. The second set of experiments was to study its efficiency in guaranteeing processing speed required by various workflow applications under different data velocity changes. The third set of experiments was to compare it with random-based immigrants GA scheme based on the proposed performance matrix. Experimental results showed that the proposed technique achieved the lowest execution cost for all workflow applications and guaranteed the speed to process incoming data streams. The quality of solution generated by the proposed technique to respond to data velocity change at runtime is high, with negligible execution time, good cost reduction and an uncompetitive number of changes required to revise the current scheduling plan. The proposed technique outperformed the competitors (baseline and genetic algorithm) to effectively respond to data velocity changes at runtime.

Chapter 6 discussed the support of additional dynamic forms of stream workflow applications (i.e. application structure changes) through a real use case, where data fluctuation is considered as one type of runtime change. These dynamic forms are modelled and based on

this model, a fully-pluggable dynamic scheduling technique was proposed to manage computing resources over time to handle structural changes of stream workflow applications at runtime while guaranteeing real-time data processing requirements. With the proposed technique, the user can easily plugin her/his elastic scheduling method to respond to runtime changes that may occur during the execution of stream workflow applications. Also, three different plugin scheduling technique were proposed, which are a baseline technique, a dynamic fair-share technique and an optimisation technique, to make scheduling decision at runtime using different heuristic approaches. The quality of response solutions generated by these techniques to handle different dynamic changes were studied through a set of experiments. The results obtained from these experiments showed that the proposed optimisation technique outperformed the other techniques and demonstrated that the complex and advanced heuristics are necessary to improve the quality of response solution.

7.2 Future Works

Studies in this thesis have opened a number of directions that could serve as a starting point for future research into stream workflow scheduling and execution in cloud environments.

The first research direction is developing an offline genetic algorithm with different advanced operators. In this thesis, we proposed a traditional genetic algorithm to schedule stream workflow at deployment time in the cloud and then improved it with a random-based immigrant schema. There are other advanced genetic operators and techniques that can be used to enhance the diversity of this algorithm such as family competition selection and replacement schemas (García-Martínez et al., 2018) (Lozano et al., 2008). Studying each of these operators will provide further understanding of the cost reduction achieved when executing stream workflow application over cloud infrastructures. This study will also help to find the best operator in order to further improve the efficiency of genetic algorithm for this workflow application.

The second research direction is workflow scheduling with fault-tolerance. The execution of stream workflow application is continuous as analytical components involved in this application are always in an active state. In the case where any VM goes down due to the volatility of cloud resources, this failure needs to be handled. Considering the distributed execution of a stream workflow over multiple cloud resources, maintaining and tracking these resources is challenging. Thus, future research on fault-tolerance techniques to handle any failure that may occur at VM level is needed.

The third research direction is developing a self-modifying heuristic technique. In Chapter 6, we presented dynamic pluggable scheduling technique that accepts user plugin heuristic methods to handle application structural changes. From this work, a new research challenge is to further assess the changing behaviour of stream workflows and execution environments, and let a self-modifying heuristic technique adopts itself based on the continuous analysis obtained to select the best heuristic from multiple plugin heuristics to apply for handling

runtime change at given time. This needs to be capable of monitoring and integrating multiple plug-in algorithms and methods to make intelligent decision based on the detailed analysis of application and runtime changes.

The fourth research direction is workflow scheduling with migration support. The aim of the migration process with a stream workflow is to move its analytical components from one cloud infrastructure to another, allowing to achieve specific real-time data processing requirements and even targeting many optimisation requirements. Examples of these optimisation requirements are performance enhancement, execution cost reduction and data locality exploitation after moving a data source to a new location. Based on this, the challenge is to achieve seamless migration without affecting the stability of workflow application and violating user-defined real-time data analysis requirements.

The fifth research direction is security-preserving scheduling. In this thesis, we utilised a Multicloud environment as an execution environment for stream workflows. Thus, the analytical components involved in this application do not reside in one cloud infrastructure, but instead are distributed over multiple cloud infrastructures for efficient execution, so that no cloud providers for these infrastructures learns or resides the whole workflow application. However, the security concerns of stream workflow applications and its big data products need to be investigated. These include preserving the security and integrity of data streams and the computation carried-out on these streams. Therefore, there is a need to design new security methods to tackle these concerns.

The sixth research direction is developing a workflow management system. Throughout this thesis, we proposed several scheduling algorithms and methods to schedule stream workflows efficiently over cloud infrastructures while maintaining real-time data processing requirements and handling runtime changes with minimal execution cost. These findings have introduced a future work in term of proposing a stream workflow management framework. This framework will incorporate the resource allocation and scheduling algorithms and techniques proposed in this thesis to handle the whole execution process of stream workflow applications in real cloud computing environments. This needs to provide the ability for user to design, deploy, monitor and manage stream workflow applications. Thus, such a framework will further assist in supporting the heterogeneity and dynamism of stream workflow for analysing IoT big data to make better decisions in real-time.

Appendices

Appendix A

Chapter 2 Appendix

A.1 Big Data Workflow Applications

In this section, we briefly overview big data workflow applications. To aid understanding of the big data workflow, we will present two additional representative examples in the subsequent paragraphs.

Big Data Workflow in the Medical Domain In precision medicine in pathology, a typical example of big data workflow is a workflow for molecular pathology in oncology, as presented in (Meyer, 2014). The representation of this workflow is depicted in Figure A.1. It is used to analyse clinical and lab data of the patient and interpret the results to improve patient care. In this workflow, the medical record is the input to the source tissue and clinical information step, which can be stored in data storage such as SQL database, while the results, i.e. clinical data including tissue of origin, history of treatment and other data, could be stored in some NoSQL databases. The sample is the input to perform clinical-grade analysis across various types of assay (analytic procedure) including Next-Generation Sequencing (NGS), array Comparative Genomic Hybridisation (aCGH), Immunohistochemistry (IHC), Fluorescence In Situ Hybridisation (FISH) (Gajecka, 2016), Cyometry and Polymerase Chain Reaction (PCR), where the results, i.e. molecular and other lab data, could be stored in some NoSQL databases. Hence, the volume of data being generated from the conducted lab tests is tremendous because of the increasing use of molecular data. Then, the results (clinical data, and molecular and other lab data) are interpreted using knowledge bases containing public, licensed, and proprietary reference content sources, where the external Application Programming Interfaces (APIs) are evoked to obtain data from those sources. Hence, heterogeneous data is being accessed as well as more data being obtained from various reference content sources for interpretation, resulting to increase in the volume of data. The results of interpretation step could be stored in some NoSQL databases. Finally, the interpreted results are used for reporting to the patient and referring clinician to produce treatment plan.

Big Data Workflow in the Business Domain In the business domain, a representative example of big data workflow is a workflow for sales conversion (Albertsson, 2016). The representation of this workflow with some extensions is shown in Figure A.2. It is used to analyse sales and page views of users to derive the conversion in order to improve sales. It differs from traditional business workflow in that there is a data analysis pipeline, data (such as page views) coming continuously as streams which are processed and analysed based on their freshness, and analytical steps such as reviewing promotion content are performed continuously based on the arrival of new data. First of all, this workflow generates the page view with demographics by processing and analysing the two data inputs, page views data (streams of data) and user data, and generates the sales with demographics by processing and analysing the two data inputs, sales data and user data; for processing page views and user

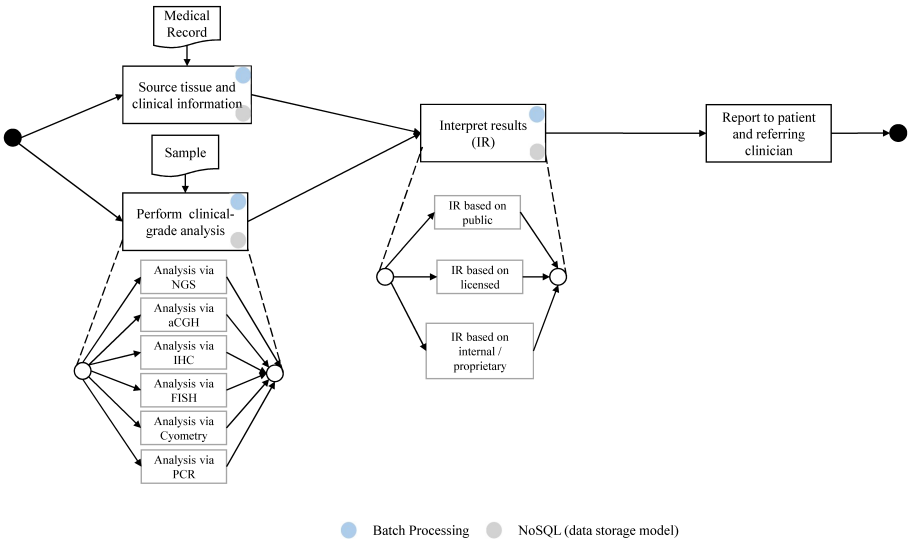


Figure A.1: An example workflow for molecular pathology in oncology

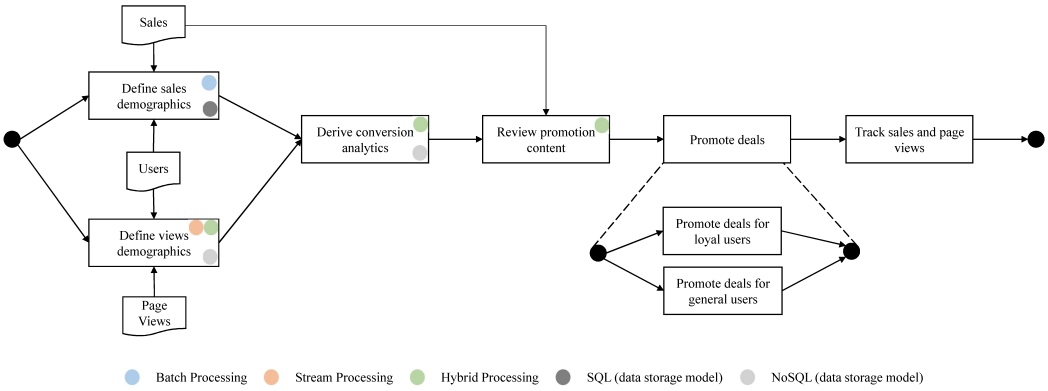


Figure A.2: An example workflow for sales conversion

data, stream data processing or hybrid data processing can be used, while for processing sales and user data, batch data processing can be used, where NoSQL databases can be used to store views' demographics and SQL databases can be used to store sales' demographics. Next, the results are processed (by using hybrid data processing) to derive conversion analytics, which could be stored in NoSQL databases. Then, the promotion content is reviewed based on conversion analytics and sales data (by using hybrid data processing). After that, the output of the review promotion content step is used to promote new deals for both loyal users and general users. Finally, the page views and sales data are tracked in order to see the reflection of new deals on them. As a result, the volume of data produced and analysed by this workflow is huge since the sales, page views and conversion data are all tremendous amounts of data, and these data are different in type. Also, page view data is coming continuously as streams based on users' usage, where those streams are being processed and analysed in the workflow based on their freshness.

Simply from the above representative examples, we can conclude that big data workflow is dynamic in nature and can include heterogeneous data analysis steps in data-intensive

pipelines. Each of these steps can receive multiple inputs and produces one or more outputs in workflow applications. In the next section, we will compare the previous types of workflows with big data workflow to understand the evolution of workflow applications until reaching big data workflow applications.

A.2 Traditional Workflows vs Big Data Workflows

There are three well-known types of workflow, which are business workflow (Reijers, 2003), scientific workflow (Barker and Van Hemert, 2007) (Ludäscher et al., 2009) and big data workflow (Ranjan et al., 2017). Table A.1 presents a comparison among the types of workflow discussed in this sub-section. In following sub-sections, we provide detailed discussion on their structure, data flow, and control flow. That will help us to understand how big data workflows have evolved from other traditional workflows (business workflow and scientific workflow).

A.2.1 Business Workflows

The workflow technology makes it possible to coordinate groups of activities; it expresses them, the associated roles that determined how they are performed, and their execution order. Thus, for any business, achieving the objectives can be done by performing its activities in certain ways. Business workflow is intended to coordinate loose-coupled business processes and services (such as booking tickets, credit checks, billing processes/services) to achieve business goals (such as competitive advantage or improved efficiency). It is a static workflow in nature since its data sources are more or less static. It focuses on control-flow patterns and events, thus its execution model is control-flow-oriented (Redlich et al., 2014) (Barker and Van Hemert, 2007) (Liu et al., 2015b). It does not include data-intensive (data analysis) pipelines and it is steered by humans or needs human interference to be executed.

Figure A.3 shows a typical example of a business workflow, which is manufacturing workflow. This workflow scenario starts when the supplier ships raw materials to a manufacturer. The manufacturer receives these materials and use them to produce new products. The new products or manufactured products are then inspected by a quality control process to ensure the quality of final products is maintained and enhanced, and any manufacturing errors are reduced or eliminated. Products that pass quality testing proceed to the following process, packing in order to be packed. After that, these packed products are stored in a warehouse system. The marketing process proceeds with information and details of products and sends them to sales people who will promote and sell products to customers, where, if the customer purchases any of these products, the salespeople confirm the order after payment has been made and send it to warehouse system for delivery. The customer also has another choice to purchase those products by using an online order service; she/he selects the products and places a new order. Following that, the total cost is calculated and the customer makes

Table A.1: A summary of comparison among different workflow types

Workflow Type	**Static / Dynamic**	**Control Flow / Data Flow**	**Pipeline**	**Characteristics**
Business Workflow	Static	Control flow	No	+ More or less static data sources + No data-intensive pipeline + Human-centric + Repeatable processes following business rules
Scientific Workflow	Static	Data flow	No	+ More or less static data sources + Based on scientific method (testing the hypotheses) + Data is variety in types + Much human intervention or steering + Distributed and heterogeneous execution environment
Big Data Workflow	Dynamic	Data flow	Yes	+ Parallel and Distributed workflow execution + Batch, stream or hybrid data + Data-intensive pipeline + Multiple inputs and outputs for an analytical task + Dynamic execution environment

the payment. After the payment has been completed successfully, the payment confirmation of such an order will be sent to the warehouse for delivery; hence the customer will receive products along with receipt. Additionally, customer service is provided to support customers before, during and after a purchase.

For implementation of business workflow, there are two main architectural approaches (Barker and Van Hemert, 2007), which are as follows:

- Service orchestration (centralised control) – It specifies an executable process that interacts with services, whether they are internal or external services, by exchanging messages. It describes the interaction of involved services at message level, and defines the control and data flow explicitly. In orchestration, a central process (coordinator) controls the participating services and coordinates the execution of operations, and those services do not know about their participation in the high-level business process. As there is a coordinator, orchestration describes the flow of process among services from the perspective of this coordinator.

- Service choreography (decentralised control) – It describes a collaborative model of interaction among a group of services that are equally treated in peer-to-peer fashion. It is not executable directly, meaning that it is executed when all involved services in the process perform their roles (Foster et al., 2006), and it is not relying on the central process, meaning that each service participating in the choreography knows exactly its role in the interaction, when to execute this role and other services (partners) who will interact with them. The focus of choreography is on the exchange of messages, where all involved services in this collaborative environment are conscious of their partners and at what time to call operations. As there is no central process, choreography tracks

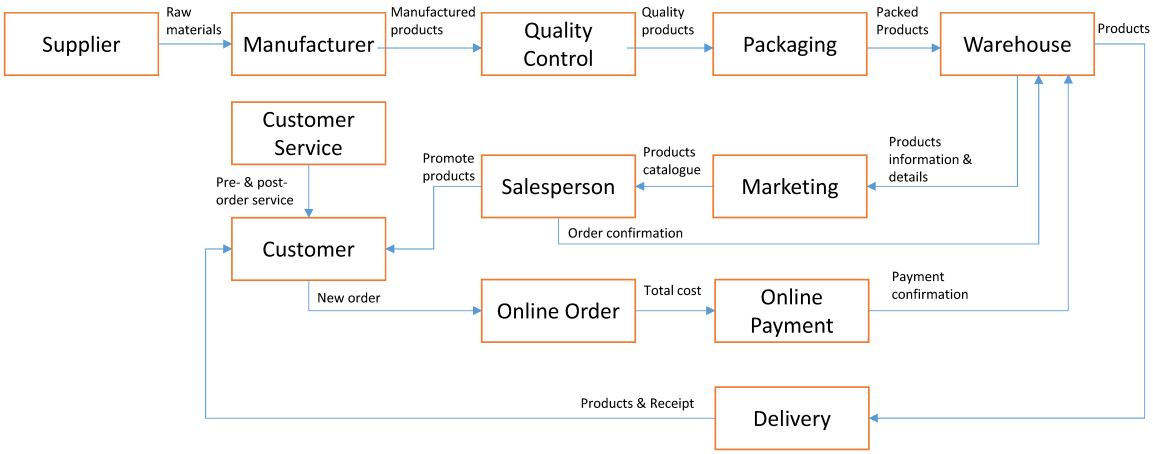


Figure A.3: A typical example of manufacturing workflow

message sequences for all participating services.

To facilitate the orchestration and choreography of a business workflow, a number of proprietary and open-source Business Workflow Management Systems (BWMSs) have been developed. Examples of these systems are Apache ODE, SAP Business Workflow, Sage CRM Workflow and ActiveBPEL. Hence, the researchers have been motivated to review/survey BWMSs and compare them with each other form different taxonomy/perspectives. Table A.2 summarises those research efforts and highlights for each research work the systems reviewed.

Since the early 2000s, business workflow has existed and it was changed after the evolution of cloud computing in 2008. This change in terms of service orchestration is in the sense that web services involved in business workflow were hosted in cloud datacenters (off-premises) instead of private hosting (on-premises hosting), while in terms of service choreography, the use of cloud or not does not make any difference because the focus is on describing a collaborative model of interaction between web services disregarding where they are hosted, but on the service hosting side, the scaling of web services becomes easy and feasible. Consequently, QoS management changed from best-effort before the emergence of cloud computing to managed after the cloud.

A.2.2 Scientific Workflows

Scientists in several domains of e-science have tried to compose various services in order to facilitate the research and discovery in their domains. For that, the workflow concepts have been applied to solve scientific problems, hence the term scientific workflow (Barker and Van Hemert, 2007). Scientific workflow allows scientists to model various computational activities included in scientific experiment in order to execute them in a specified order (Liu et al., 2014). In other words, it allows composition of a series of data processing/manipulation steps, which are illustrated in the computational experiment design process. While the business workflow is intended for the business community, the scientific workflow is intended

Table A.2: Review and survey research works for BWMSs

Research Work	**Summary (for BWMSs part and related part)**	**Reviewed Systems**
(Harrer et al., 2014)	This work proposed a pairwise approach that relied on BPEL Engine Test System (betsy), and then evaluated static analysis conformance of open-source BPEL-based systems by applying the proposed approach. It also presented the obtained results and discussed these results.	ActiveBPEL bpel-g Apache ODE OpenESB Orchestra Petals ESB
(Harrer et al., 2013)	This work reviewed and compared proprietary and open-source BPEL-based systems by the use of the proposed testing approach (which relied on the extended version of automated testing tool betsy), which evaluates those systems in term of standard conformance and language expressiveness.	P1 (name undisclosed) P2 (name undisclosed) P3 (name undisclosed) bpel-g Apache ODE OpenESB Orchestra Petals ESB
(Harrer et al., 2012)	This work reviewed and examined open-source BPEL-based systems by utilising the automated testing tool betsy, which evaluates the standard conformance of those systems.	bpel-g Apache ODE OpenESB Orchestra Petals ESB
(Garcês et al., 2009)	This work reviewed and evaluated 10 different open-source workflow management systems using the proposed framework that focuses on the compliance of those systems with a Workflow Management Coalition (WfMC) reference model, as well as on runtime and design time perspectives, to provide decision makers a starting line for choosing the workflow management solution.	Bonita Enhydra Shark JawFlow JBoss jBPM JFolder JOpera OpenWFE RUNA WFE WfMOpen YAWL
(Mans et al., 2009)	This work evaluated four workflow management systems based on the flexibility requirements needed for supporting organisational healthcare processes.	ADEPT1 YAWL FLOWer DECLARE

for the scientific community and has its own domain specific QoS and SLA requirements (Barker and Van Hemert, 2007).

Scientific workflow is a static workflow in nature since its data sources are more or less static. It is positioned around performing scientific experiments and automating these experiments in a distributed manner, so that its elements enabled the scientist to prove scientific hypotheses. The process of building such workflow in order to accomplish the validation of scientific hypotheses will commonly be constructed incrementally (not designed and then implemented). The execution model of scientific workflow should be dataflow-oriented due to proving the scientific hypotheses relying on empirical data (Liu et al., 2015b) (Barker and Van Hemert, 2007).

Scientific workflow consists of scientific data manipulation activities and dependencies

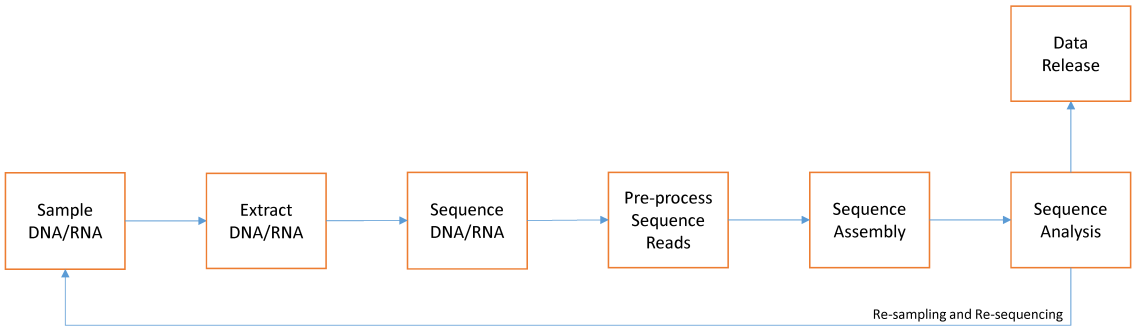


Figure A.4: A typical example of genomics workflow

within them. The data manipulation activity is one logical data processing step within the representation of scientific workflow, and data dependency is a relation between two activities, where the output data of input activity is used by output activity. During the execution of scientific workflow, one data manipulation activity might contain numerous executable tasks for various portions of experimental data, where an individual task is intended for processing the input data portion of such activity (Liu et al., 2014). The type of data being injected and processed in such workflows is varying in type. The execution of such workflow is a greatly steered by humans and is carried-out in various distributed execution environments. A typical example of scientific workflow is genomics workflow (Cofer et al., 2015). The representation of this workflow is depicted in Figure A.4, illustrating the process of workflow starting from initial sampling on the way to final results. This workflow scenario consists of five activities (sampling Deoxyribonucleic Acid (DNA)/Ribonucleic acid (RNA), extracting DNA/RNA, sequencing DNA/RNA, pre-processing sequence reads, assembling sequence and analysing sequence), and the relations between those activities are data dependencies.

To facilitate the choreography and orchestration of scientific workflow, numerous Scientific Workflow Management Systems (SWMSs) have been developed in the last 20 years. They are intended for managing scientific workflow modelling, execution and datasets in different execution environments such as cluster, grid and cloud (Barker and Van Hemert, 2007).

In the scientific domain, various languages and systems that have been developed provide the ability for scientists to automate the computational tasks of an experiment through a workflow, which makes the SWMS offering diverse including Taverna, Pegasus, Kepler, Triana and Swift. Since many SMWSs have been developed, the researchers have been motivated to review as many systems as possible, and analyse them or present a comprehensive comparison among them. As a result, a lot of research work has been published that surveys and taxonomies scientific workflow management systems. For simplicity, we present those research efforts in Table A.3 and highlight for each research work the systems reviewed and research taxonomy/perspectives used in comparison if it is conducted.

The advances in scientific experiments lead to those experiments that handle and consume tremendous volumes of data and their data processing includes several data analysis

tasks and dependencies between them, resulting in the scientific workflow becoming more data-intensive, defined as data-intensive scientific workflows. The data-intensive scientific workflow is a viable solution for modelling large-scale scientific problems (Liu et al., 2015b). Hence, to meet the requirements for data-intensive scientific experiments, SWMSs integrate the techniques of data parallelism and exploit the distributed sources offered by various execution environments such as cluster and cloud (Liu et al., 2015b) (Wang et al., 2014a). These workflows are the preliminary form of new types of workflows that are emerging with big data platforms and analytical technologies. However, there are several research works have been proposed scientific workflow systems with big data extensions. These efforts are summarised and compared in Table A.4.

A.2.3 Big Data Workflows

From the time of introducing scientific workflow, and the demand for solving data computation problems that are more complex increasing, big data workflow is becoming gradually viable. It is clear that several existing workflow systems such as Nova, Oozie, have extended their functionalities to support programmability of MapReduce applications (Wang et al., 2014a). In contrast to business and scientific workflows, the big data workflow supports composition of heterogeneous big data analytics applications (e.g. batch processing, stream processing, SQL, NoSQL, Ingestion) into a holistic data processing pipeline to perform complex data computation operation over high volume, high variety, and high velocity data (Ranjan et al., 2017). Simply, big data workflow differs from the previous types (business and scientific) of workflows in that it is significantly more complex, dynamic, and heterogeneous: data comes in different shapes, at different volumes and at different speeds. Consequently, the execution and managing of big data workflow needs a dynamic environment that provides the underlying infrastructure for big data processing, allows parallel execution of such workflow and exploits huge amounts of distributed resources. Cloud provides on-demand access to infinite resources virtually including compute, storage and network (Liu et al., 2015b), which offers great convenience for processing massive amount of data (Zhao et al., 2014). As a result, the dynamic nature of big data workflow allows several big data applications to process and analyse dynamic datasets to achieve certain goals and the dynamic nature of cloud computing enables running of composed applications over a highly dynamic environment in a parallel and distributed manner.

A.3 Big Data Application Orchestrator

Table A.5 highlights the capabilities of each one of them.

Apache YARN Apache YARN (Vavilapalli et al., 2013) (Apache, 2017) is a cluster management technology/platform that aimd at spliting-up the functions of resource management

Table A.3: Review and survey research works for SWMSs

Research Work	**Summary (for SWMSs part and related part)**	**Reviewed Systems**	**Research Taxonomy / Perspectives**
(da Silva et al., 2017)	This work reviewed 15 selected workflow management systems and provided a comparison among them using features that are related with extreme-scale applications as well as discussed future research challenges for achieving the demands of these applications	ADIOS, Apache Airavata, Askalon, Bobolang, dispel4py, Fireworks, Galaxy, Kepler, Makeflow, Moteur, Nextflow, Pegasus, Swift, Taverna, Triana	workflow execution models, heterogeneous computing environments and data access methods
(Liu et al., 2015b)	This work reviewed the existing SWMSs and provided a comparison among them, and discussed the execution of data-intensive scientific workflows in parallel over various execution environments, specifically in the cloud	Pegasus, Swift, Kepler, Taverna, Chiron, Galaxy, Triana, Askalon, WPg	Key focus on workflow structure, user interface, scheduling and parallelism
(Bux and Leser, 2013)	This work presented a taxonomy of important parallelisation aspects for SWMSs, a comparison among parallelisation techniques and a discussion for the current SWMS running on computational infrastructures as well as outlined current trends and research questions	Swift, Condor, DAGMan, Pegasus, Taverna, KNIME, Kepler	SWMS's features and applications
(Talia, 2013)	This work discussed scientific workflow concepts and reviewed 10 SWMSs as well as argued some open and research issues.	Taverna, Triana, Pegasus, Kepler, Askalon, Weka4WS, GWES, DVega, Karajan, DIS3GNO	No comparison
(Curcin and Ghanem, 2008)	This work reviewed four scientific workflow systems in addition to YAWL and BPEL and conducted a comparison between those systems based on some syntactical features, and their data-flow and control-flow properties	Discovery Net, Taverna, Triana, Kepler	Workflow representation based on control and data flow

continued ...

. . . continued

Research Work	**Summary (for SWMSs part and related part)**	**Reviewed Systems**	**Research Taxonomy / Perspectives**
(Zhao et al., 2008)	This work determined the challenges to scientific workflow development, outlined some existing and emerging workflow technologies, and discussed opportunities to address these challenges	DAGMan , Pegasus, Taverna, Kepler, Triana, Workflow Bus project, WWF, Star-P, Swift	No comparison
(Barker and Van Hemert, 2007)	This work surveyed existing workflow technology from two different domains (i.e. business domain and scientific domain), reviewed the most popular scientific workflow systems and presented a discussion for improvements of workflow systems	Taverna, Kepler, Triana, Pegasus, GridNexus, DiscoveryNet	No comparison
(Zhao et al., 2005)	This work presented the vision of the authors on scientific workflows and SWMSs and reviewed thirteen SWMSs from the viewpoints of workflow model, execution and runtime model, and user support	Askalon, GEODISE, DAGMan , GridAnt, GridBus, Grid-Flow, GridNexus, ICENI, Kepler, Pegasus, SPA, Taverna, Triana	Workflow models including execution and run time models with user support
(Bhagwanani, 2005)	This work presented approaches for designing of user interfaces of SWMSs, and usability heuristics and metrics that are used to compare the five reviewed SWMSs	Pilot system, Ptolemy II based SPA, SCIRun Enhydra JaWE, ObjectWeb Bonita, Taverna	Focused on usability aspects
(Yu and Buyya, 2005)	This work presented a taxonomy for scientific workflow systems in grid environments, and reviewed grid workflow systems and compared them using the proposed taxonomy	DAGMan , Pegasus, Triana, ICENI, Taverna, GrADS, GridFlow, UNICORE, Gridbus workflow, Askalon, Karajan, Kepler	Workflow Design, Workflow Scheduling, Fault-tolerance, Data movement

Table A.4: Comparison between the reviewed research works (data-intensive scientific workflow management systems)

Ref.	SWMS Used	Big Data Platform and Tool	Approach	Data Model	DDP Pattern	Cloud Environment	Use Case (Workflow) / Real Application
(Filgueira et al., 2016)	Pegasus and dispel4py	Apache Storm, MPI, Multiprocessing and Sequential	dispel4py workflow modeling	Stream	dispel4py processing element	Supported	Seismic Ambient Noise Cross-Correlation (its representation run on academic NSF-Chamelon cloud
(Wang et al., 2014a)	Kepler	Hadoop and Stratosphere	Actor-oriented modeling paradigm	Batch	Map, Reduce, CoGroup, Match and Cross	Not supported	RAMMCAP and CloudBurst (Bioinformatics applications)
(Kashlev and Lu, 2014)	VIEW	-	Scientific Workflow Language (SWL) specification	-	Workflow construct	Supported	Big data workflow - Analysing Driving Competency from the Vehicle Data
(Wang et al., 2009b)	Kepler	Hadoop	Actor-oriented modeling paradigm	Batch	Map and Reduce	Not supported	Word count

Table A.5: The main capabilities of big data workflow systems

Orchestrator	**Scheduling**	**Fault-tolerance and High Availability**	**Security**	**Monitoring**
Apache YARN	Monolithic scheduler (single centralised scheduler)	-Fault-tolerance at each layer of YARN stack -RM's HA (manual transition and failover, and automatic failover)	-Authentication (Kerberos authentication) -Authorisation (ACLs and Service level authorisation) -Data confidentiality (SSL and HTTPS)	RM Web user interface and NM Web user interface
Apache Mesos	Dual-level scheduling (Mesos - low-level infrastructure scheduler and Framework - application specific logic scheduler)	-ZooKeeper for fault-tolerance -Master's HA (automatic recovery)	-Authentication (Cyrus SASL uses CRAM-MD5 or custom module) -Authorisation (ACLs and roles) -Data confidentiality (SSL and HTTPS)	URL for observability metrics and per container network monitoring and isolation
AWS Lambda	-	-Built-in fault-tolerance -High availability by design	-AWS SDK and AWS IAM (fine-grained access control, AWS MFA and federated access) - VPC for Lambda function	AWS Lambda console, CloudWatch console/CLI/API and AWS CLI

and job life-cycle management (i.e. JobTracker responsibilities). It has three core compoenents, which are NodeManager, ResourceManager (RM) and ApplicationMaster (AM).

The TaskTracker is replaced by NodeManager (NM), which is per-machine framework agent that is in charge for containers of applications, monitoring their usage of typical resources, and detailing the same to the RM. The RM is a central authority that is responsible for managing and tracking the cluster resources, and mapping them to competing applications in the system and is composed of two main components, which are Scheduler (to map resources to applications according to familiar constraints of capacities) and ApplicationsManager (to accept submissions of job and negotiate the execution of the application specific ApplicationMaster for a container as well as offer the ability to restart the AM container in case of failure). The RM and NM form a new system, data-computation fabric that aimed at managing applications, where an application could be either a single job or a network of jobs (expressed in directed acyclic graph). The AM as a framework specific library is responsible for managing the scheduling and coordination of applications (i.e. dynamically negotiating and accessing resources from the RM as well as talking with the NMs in order to start, track and monitor the applications' tasks). The main capabilities of Apache YARN can be summarised in four aspects, which are scheduling, fault-tolerance and high availability, security, and monitoring. These aspects are outlined in the below paragraphs.

Scheduling. Apache YARN uses a monolithic scheduler to map compute resources among competing applications in the cluster. When the job is submitted, it is received by RM who is evaluating the available resources and making the decision where such job should go (Scott, 2015). The RM uses its two parts, Scheduler and ApplicationsManager, to take care of that. It uses ApplicationsManager to accept application/job submission and starts application specific AM (Lynn, 2016); it uses a pluggable Scheduler that is either Capacity Scheduler, which allow for multi-tenant Hadoop applications to securely share a large cluster while maximising cluster utilisation and throughput; or it uses Fair Scheduler, which allows for Apache YARN applications to fairly share compute resources in large clusters (Apache, 2017). Moreover, Apache YARN was designed for optimising the scheduling of Hadoop jobs (i.e. batch jobs), but not for services with long runtimes, such as SOA applications, nor for short-lived interactive queries (real-time workloads), such as stream jobs, and "while it's possible to have it schedule other kinds of workloads, this is not an ideal model" (Scott, 2015). It also was not designed for stateful services, instead it is for stateless batch jobs that can be restarted easily in case of failures (Scott, 2015). Furthermore, it does not support workflows and the dynamic composition of data-intensive activities. In addition, since the evolve of Apache YARN's monolithic scheduler to handle various workload types is theoretically possible by combining new scheduling algorithms upstream into the code of scheduling, this is not a lightweight model if taking into consideration the growing number of present and future scheduling algorithms that they need to support (Scott, 2015).

Fault-Tolerance and High Availability. Fault-tolerance is built into every layer of Apache YARN stack, so that the complexity of detecting hardware faults and recovering from them is hidden from users, and the responsibility is now distributed among RM and AMs operating in the cluster system (Vavilapalli et al., 2013). When RM fails, the RM itself starts its recovery process to recover from its own failures by getting back to its state from a persistent store on initialisation (Vavilapalli et al., 2013), where RM restart feature allows it to keep functioning across restarts with one of two restart options: non-work-preserving RM restart (focus on persisting the state of application/attempt and other credentials information in a pluggable state-store) and work-preserving RM restart (focus on reconstructing the RM running state) (Apache, 2017). While in the case of the failure of NM, the RM detects such failure by timing-out its heartbeat response and restarts it. As well, when AM fails, the RM may restart it. Of course, handling the failures of containers is the responsibility of the frameworks (Vavilapalli et al., 2013). For high availability, Apache YARN's RM is relied on Active/Standby architecture, where one RM is active while one or more RMs are remaining in standby mode and they are ready to take over in case of something goes wrong to the active such as it goes down (Apache, 2017). The transition-to-active can be triggered manually by admin (i.e. manual transition/recovery) using a command line interface if automatic-failover is disabled or automatically (i.e. automatic transition/recovery) via a Zookeeper-based ActiveStandbyElector embedded in the RM if automatic-failover is enabled (Apache, 2017) (Lynn, 2016). Thus, the separate ZooKeeper Failover Controller is not needed since the RMs embed the ActiveStandbyElector that performs as a failure detector and a leader elector (Lynn, 2016). The usage of ZooKeeper is to record RMs state (Lynn, 2016).

Security. Apache YARN supports authentication, authorisation and data confidentiality to enforce security in the system (Lynn, 2016). The Kerberos authentication is used to authenticate each user and service. Access Control Lists (ACLs) is used to finely control the access to the Hadoop services, and service level authorisation is used to ensure that the Hadoop services are used only by users who have rights to use them. Moreover, the Secure Sockets Layer (SSL) is used to encrypt data and communication between users and services, and HTTPS ensures that the data transferred between Web console and the users in secure connection (Lynn, 2016). Additionally, in the context of secured Apache YARN clusters, Apache YARN supports secure containers (Apache, 2017). The containers uses the facilities of the operating system to offer execution isolation for containers, where the execution of secure containers is done under the job user credentials. The access restriction for the container is enforced by such an operating system as well as the running of the container should be the same as the user that submitted the application (Apache, 2017).

Monitoring. Apache YARN provides Web user interface for both RM and NM, where RM user interface offers metrics for Apache YARN cluster while NM user interface provides for each node in the cluster, information about such node, and the applications and containers

running on this node (Lynn, 2016).

Apache Mesos Apache Mesos (Hindman et al., 2011) (Sphere, 2017a) is an open-source platform that abstracts the entire datacentre into a single pool of computing resources to allow for efficiently sharing those resources among various cluster computing frameworks such as Hadoop, Spark, Message Passing Interface (MPI). The two main design principles that Mesos is built around are a distributed scheduling mechanism called resource offers that delegates the responsibility of the decisions of scheduling to applications (called frameworks in Mesos), and a fine-grained sharing model at tasks level (Hindman et al., 2011). Mesos architecture is a dual-level architecture (or two-level scheduler architecture), where the scheduling operations of the low-level infrastructure are handled by Mesos while all the application specific logic is handled by the framework. With this architecture, Mesos is capable of supporting different kinds of workloads (e.g. service, batch or streaming), scaling to a large number of nodes as well as extending to support new distributed technologies (Sphere, 2017a). Mesos comprises of two main components. The first component, master process is to manage Mesos agent daemons that are running on every cluster node, and the second component, frameworks is to run applications' tasks on these agents (Hindman et al., 2011). Each framework running on Mesos comprises of scheduler and executer process, the scheduler is responsible for registering with a master to get resource offers and the executor is started on agent nodes for executing the tasks of framework (Hindman et al., 2011).

The main capabilities of Apache Mesos can be summarised in four aspects, which are scheduling, fault-tolerance and high availability, security, and monitoring. These aspects are outlined in the below paragraphs.

Scheduling. The master process in Apache Mesos implements fine-grained sharing model across frameworks by the use of "resource offers". The "resource offers" scheduling mechanism makes offers of resources to a framework and let this framework either accept the offer, or reject it if the offered resources do not meet its constraints and then waiting for ones they do (Sphere, 2017a) (Hindman et al., 2011). Thus, the master is responsible for deciding how many resources that can be offered for each framework in accords to an allocation policy (e.g. priority or fair sharing) defined by administrator via a pluggable allocation module, while frameworks take the responsibility for deciding which offered resources to be accepted as well as which workloads to be executed on these resources (Hindman et al., 2011). Moreover, in Apache Mesos, Metronome is a job scheduler for scheduling DC/OS jobs, and on top of Apache Mesos, there are two service schedulers, Apache Aurora for running stateless services that written in any language and Marathon for running containers (including Docker). Additionally, Apache Mesos allows both coarse-grained control and fine-grained control of resources in a system, and that's can be in the same cluster, where some applications can use coarse-grained control while others use fine-grained control (Lynn, 2016).

Fault-Tolerance and High Availability. Apache Mesos makes its master fault-tolerance by supporting automatic recovery of the master using Apache ZooKeeper, where the currently running tasks can continue to execute in the case of failover (Lynn, 2016). The Mesos master is designed to become soft-state, which means reconstruction the internal state of a new master can be done completely from information held by agent nodes and the framework schedulers (i.e. list of active agent nodes, active frameworks and executing tasks) (Hindman et al., 2011). While, in the case of the failures of node and executor, Mesos reports those failures to the schedulers of frameworks, and then these frameworks take the appropriate actions to react to those failures in accordance to their policies (Hindman et al., 2011). For high availability, multiple masters are running in a hot-standby configuration using ZooKeeper for leader election, so that if the master fails, another master is chosen as a leader, and agent nodes and frameworks connect to this leader and repopulate its state (Hindman et al., 2011). Moreover, for the failures of framework schedulers, Mesos allows a framework to register multiple schedulers with master, so that in case of one of them fails, the master notifies another one to take over (Hindman et al., 2011).

Security. Apache Mesos supports authentication, authorisation and data confidentiality to enforce security in the system. In Apache Mesos, only trusted entities can interact with the cluster, so that it is required for any entity to be authenticated to interact with the cluster, and that's including frameworks registering with the master, agents (node slaves) registering with the master and operators using endpoints such as HTTP endpoints (Sphere, 2017b) (Lynn, 2016). By default, authentication is disabled and when it is enabled, the operators can configure Apache Mesos to use either default authentication module (i.e. Cyrus SASL that uses CRAM-MD5 as a default authentication method), or custom authentication module (Sphere, 2017b). For authorisation, ACLs are used to authorise access to services in Apache Mesos, and roles, on the other hand, are used to associate certain resources with frameworks (Lynn, 2016). The flow of all messages between Mesos components in the cluster are unencrypted by default, thus SSL/TLS is supported for enabling and enforcing SSL communication to encrypt low-level communication. As well, the support of HTTPS is added to the Mesos WebUI (Sphere, 2017b).

Monitoring. Apache Mesos provides the observability metrics for Mesos master and agent nodes accessible via a Uniform Resource Locator (URL) (Sphere, 2017b) (Lynn, 2016). The two types of metrics provided are counter and gauge, where the counter metric keeps the track of discrete events and is increasing monotonically, for example, number of failed tasks or number of valid framework messages, while gauge metric represents an instant sample of some magnitude, for example, number of connected agents or percentage of allocated memory (Sphere, 2017b). Moreover, Apache Mesos supports per container network monitoring and isolation (Lynn, 2016).

AWS Lambda AWS Lambda (Amazon, 2017b) is a compute platform/service that provides the ability to run code written in Java, C#, Python or Node.js, all without the need to provision and manage compute resources and with zero administration. The code uploaded to AWS Lambda is called a "Lambda function", where the execution of this function on Lambda resources is in response to events, for example, actions by users, changes in system state or changes in data. Thus, by using this service, various data processing systems can be built such as real-time stream processing or Extract, Transfer, Load (ETL) systems. Lambda function is stateless with no convergence to the underlying compute resources, thus multiple copies of the function can be rapidly launched as needed by Lambda in order to scale to the incoming events rate. AWS Lambda allows running the function at Edge locations using Lambda@Edge, so this function will run at global AWS edge locations without the need to provision and manage the underlying compute resources, and in response to end users at the lowest network latency. AWS Lambda charges for the number of requests for Lambda function and the compute time this function executes.

The main capabilities of AWS Lambda can be summarised in three aspects, which are fault-tolerance and high availability, security, and monitoring. These aspects are outlined in the below paragraphs.

Fault-Tolerance and High Availability. AWS Lambda embed fault-tolerance for maintaining compute capacity across various Availability Zones in each region, which assists in conserving the function in case of failures. As well, the predictable and reliable operational performance is provided by AWS Lambda and Lambda functions that operating on the service. For high availability, the design of AWS Lambda allow supporting that, for the service and the Lambda functions it executes.

Security. AWS Lambda has built-in AWS Software Development Kit (SDK) and integrates with AWS Identity and Access Management (IAM) in order to provide the ability for the code/function to securely access other AWS services and resources. AWS IAM allows to use fine-grained access control (for controlling users access to AWS resources), to use AWS Multi-Factor Authentication (MFA) (for highly privileged users), and to integrate with existing identity system such as organisation directory (for granting federated access to AWS service APIs or management console). By default, AWS Lambda runs the code/function within a Virtual Private Cloud (VPC); as well, it can be configured to allow access to resources behind a VPC.

Monitoring. AWS Lambda monitors Lambda functions automatically and reports metrics through Amazon CloudWatch, which include total requests, latency and the rates of error. Accessing to these metrics and viewing logs is by using AWS Lambda console, CloudWatch console/CLI/API and AWS CLI.

A.4 Data Security and Privacy

In cloud computing, data security remains a major concern. When the execution of big data workflow moves to the cloud, big data being processed by workflow tasks will be stored in and accessed from the cloud, which poses challenges of securing big data. These challenges are securing such data, ensuring its integrity, availability and privacy, and controlling access rights.

Data Confidentiality, Integrity and Availability The use of data encryption allows us to preserve data confidentiality. The encryption and decryption of huge datasets are non-trivial tasks because they are computationally expensive and time-consuming as well as the fact that performing them during the execution of workflow means suspending the execution of some workflow tasks and waiting until the encryption/decryption process completes. Data integrity as a one main aspect of big data security aims to ensure the consistency and accuracy of the data before and after computational operation. Ensuring the correctness of large datasets being processed by workflow tasks is challenging because of the diversity of sources and heterogeneous nature of data flow. Moreover, execution of big data workflow across multiple cloud resource types (virtual machines/CPUs, storage, etc.) further complicates the data confidentiality, integrity, and availability problems (due to the fact that such datasets may be stored in various cloud datacenters) (Saha and Srivastava, 2014). Data availability is another important aspect, which aims to make data available when an authorised user requests it (Krutz et al., 2010). As the large dataset being processed by a workflow is outsourced to the cloud and stored in cloud storage, the responsibility of ensuring data availability will turn to cloud providers; thus the challenge here is the decision to choose a cloud storage service(s) that guarantees high availability of data. For that, the need to craft a balance between the security of big data and its availability to gain the benefits of both without the loss of any part of them is important. Consequently, the aspects of big data and the diversity of sources for such data complicate achieving data confidentiality, integrity and availability.

Access Control/Authorisation Another key aspect of big data security is enforcing restriction on data access. The characteristics of cloud computing as an execution environment for big data workflow make enforcing access control a challenging task. The datasets may be stored in various cloud storage services or systems and accessing such data varies from provider to provider, thus this task becomes more difficult.

Privacy Big data is an asset and may contain private data besides the hidden insights, so that outsourcing such data to be processed on the cloud raises the challenge of preserving data privacy. This challenge is non-trivial because ensuring privacy not only needed for working datasets, but also for intermediate and final results. It becomes more difficult when

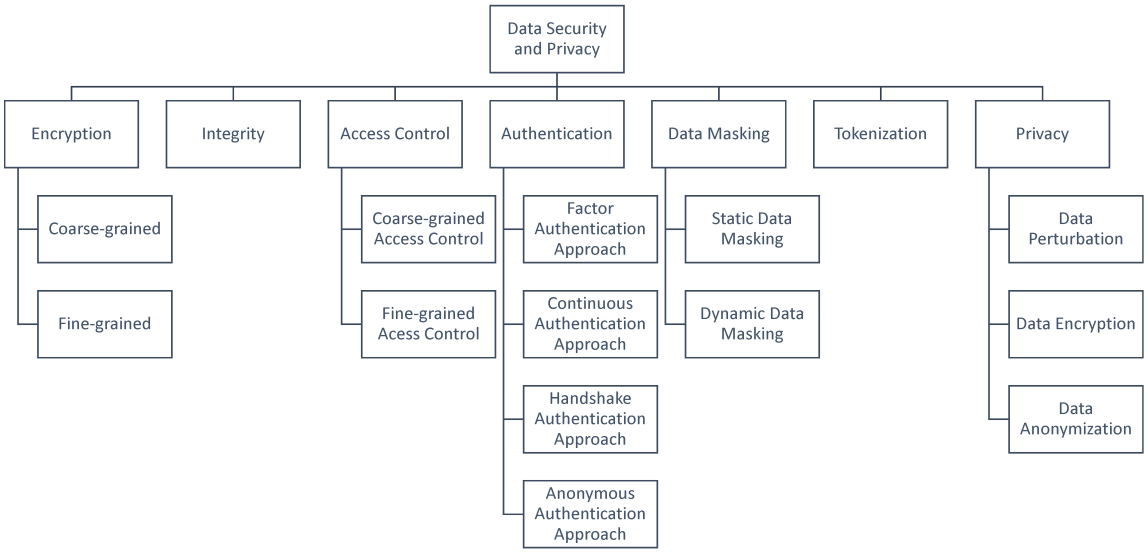


Figure A.5: Classification of Data Security and Privacy Approaches for Workflow

personal information or private data is required for relevant results.

Accordingly, preserving the security and privacy of data is significant to protect both data and identity, and to enforce access controls. Data-centric security aims to apply security protection to the data itself rather than the security of systems. It protects data with different granularity levels in order to control as well as monitor the access to data as needed and all of that is devoid of the impact of business systems (Mattsson, 2016). In data-centric security, several security methods need to be considered to preserve data security and privacy. Figure A.5 shows the classification of data security and privacy for workflow. Table A.6 shows the comparison between different security methods.

Encryption It aims to encode data into unreadable or encrypted form, where unauthorised parties cannot easily understand it, while authorised parties only can access it. The two following encryption approaches are used in the literature (Mattsson, 2016):

- Coarse-grained encryption − This approach intends to secure actual data at coarse-grained level, which can be either extremely coarse-grained such as disk/volume level or less coarse-grained such as container level or file level. For extreme coarse-grained level, the cloud storage provider offers capability to encrypt data at storage level, mostly it is set by default, while with less coarse-grained level, the data is encrypted in a more granulated way, which is mostly carried-out by users such as encrypting a set of files or datasets residing cloud storage. This approach at extremely coarse-grained level provides security to data-at-rest, but the plaintext data is required for data-in-transit, data-in-use and data-in-analysis states. Thus, the privileged users are still able to access sensitive data (Mattsson, 2016).

- Fine-grained encryption − The level of encryption in this approach is more detailed. Thus, this approach allows adding strong encryption in the precise level (such as for

Table A.6: Evaluation of security methods based on data states and protection from privileged users

Security Method	**States of Data Security**				**Protection from Privileged Users**
	Data-at-rest	**Data-in-transit**	**Data-in-use**	**Data-in-analysis**	
Access Control	No	No	Yes	Yes	Low/Little
Authentication	No	No	Yes	Yes	Low/Little
Coarse-grained Encryption	Yes	No (at extremely coarse-grained level) Yes (at less coarse-grained level)	No	No	Very little (at extremely coarse-grained level) Moderate (at less coarse-grained level)
Fine-grained Encryption	Yes	Yes	No (some improvements preserve some security at this state)	No (some improvements preserve some security at this state – format-preserving security)	High/Much
Static Data Masking	Yes	Yes	Yes	Yes	High/Much
Dynamic Data Masking	No	No	Yes	Yes	Low/Little
Tokenisation	Yes	Yes	Yes	Yes	-

fields or columns) to provide more security protecting data-at-rest and data-in-transit, and more protection from privileged users. During the execution of job functions including analysis, the data is required to be in plaintext format so that there is no security protection to data-in-use and data-in-analysis (Mattsson, 2016). The format-preserving encryption can provides the ability for users as well as applications to access the protected data, however it is considered as a one of the most slowly performing encryption.

Integrity It aims to control the lifecycle of data to ensure its consistency and accuracy. The data integrity approaches can be classified into two approaches based on the need to retrieve data itself for checking (Liu et al., 2015a). These approaches are as follows:

- Data Retrieval-Based Approach − This approach requires to download or retrieve data in order to verify its integrity. Digital signature schemas such as RSA and BLS signatures are examples of data security techniques that based on this approach.
- Non Data Retrieval-based Approach − The aim of this approach is to verify data integrity without the need of retrieving data itself to achieve that. Proofs of retrievability and provable data possession are data verification techniques based on this approach.

Access Control − This approach aimed at enforcing restriction on access to resources and allow only authorised parties to access resources that they have access rights to. During

job functions, the data is in clear so that no security is provided to data-at-rest and data-in-transit as well as not much protection from privileged users (Mattsson, 2016). The access control can be classified into two approaches based on granularity level. These approaches are as follows:

- Coarse-grained access control − It describes only the permissions to deny and restrict access.
- Fine-grained access control − It describes precisely the permissions in order to enforce detailed restriction of access. For instance, Usage Control is a fine-grained access control model that enforces fine-grained access on data by introducing access rights based on attributes (Gouglidis and Mavridis, 2009).

Authentication This approach is to verify the identity of someone or something to be valid in order to give access to confidential data or resources. For securing access to the data, there are four approaches:

- Factor Authentication Approach/Layered Authentication Approach
  - Single-factor authentication − It authenticates the identity of an individual who wants to access data through one authentication factor, i.e. knowledge factor such as password.
  - Two-factor authentication − It uses two authentication factors, i.e. knowledge and possession factors such as password and security token, to verify the identity of an individual who wants to access data.
  - Multi-factor authentication − It uses multiple authentication factors (knowledge, possession and inheritance factors) to verify the identity of an individual who wants to access data.
- Continuous Authentication Approach − It aims to continuously monitor and verify the identity of users to provide more security.
- Handshake Authentication Approach − It aims to authenticate the user by applying one or more handshake steps. One-way, two-way, three-way and four-way handshake are existing schemas based on this approach. Moreover, handshake schema can be intended to support specific-domain communication for authenticating user at this domain only, which limits its deployment in real-world scenario. While, cross-domain handshake schema (He et al., 2018) is intended to support cross-domain communication for authenticating users registered in various domains and create a secure channel.
- Anonymous Authentication Approach − It uses anonymous communication to authenticate the user for protecting her/his privacy, so the access to this user is therefore approved without disclosing the her/his identity. Anonymous authentication protocols can be for single-server architecture or multi-server architectures (He et al., 2016).

Data Masking It replaces the original data with a worthless value of the same data type and length, where the masked data looks and acts like the original, allowing users and processes to read it (Mattsson, 2016). The process of hiding data with this method is irreversible. The two methods of data masking are as follows :

- Static data masking − With this method, the sensitive values of data are replaced permanently with non-sensitive values (worthless values). It secures data in states and from privileged users. Since the masked data is irreversible, static data masking is utilised in non-production environments such as development environments and is usually not used in production environments (Mattsson, 2016).
- Dynamic data masking − In this method, the masking process is performed on the fly where the masked data is replaced by the sensitive data, which is requested by the user or process who/that does not have permission associated with the assigned role to see such data in its clear format (Mattsson, 2016). It provides security to data-in-use and data-in-analysis, and no security to data-in-transit, data-at-rest and little from privileged users. In production analytic scenarios, working with dynamically masked values could be problematic depending on the method used (Mattsson, 2016).

Tokenisation It replaces data in plaintext format with a random value of the same data format (Mattsson, 2016). The process of masking data with this method is reversible because of the use of token tables instead of cryptographic algorithms. The data being tokenised with this method can be utilised in data processing and analytics in place of cleartext data since a one-to-one relationship with the original data can be preserved. As well, in some cases where merely a portion of the original data is needed to carry-out a job, it is useful to tokenise portions of cleartext data. This method provides flexibility in the description of data security privileges (on partial field or field by field basis) and security to all data states (Mattsson, 2016).

Data Privacy It aims to protect sensitive data that the owners of that data need to be undisclosed, such as the data itself and its representation characteristics (Fang et al., 2017). There are three methods to preserve privacy. These methods are as follows (Fang et al., 2017):

- Data Perturbation − This method perturbs the original dataset to produce a noisy dataset by performing a series of operations that replace sensitive data in this dataset with perturbed data through anonymous perturbation, adding random variables, adding a random offset value, replacement, and releasing only such noisy datasets. With this method, the leakage of sensitive data is not totally prevented.
- Data Encryption − This method uses encryption technology for encoding sensitive data to assure data authenticity, reversibility, and non-destructiveness, as well as has a high degree of preserving privacy.

- Data Anonymisation − This method attains privacy protection by hiding user identity and sensitive data (Fang et al., 2017). It converts the original data to anonymised data that makes it is impossible or at least very difficult to disclose the identity or sensitive data. It is finding the middle ground between the risk of privacy disclosure and the precision of data, and releasing sensitive data in a selective manner.

Appendix B

Chapter 4 Appendix

B.1 Relative Difference of Execution Cost Results of Scenario 1 and 5

Figure B.1 and Figure B.2 show the experimental results of Scenario 1 and 5 for the relative difference (in percentage) that achieved by the proposed algorithms and fair-share algorithm in comparison to lower bound in term of execution cost.

B.2 Average Latency Results of Scenario 1, 5 and 6

Figure B.3 to Figure B.5 show the experimental results of Scenario 1, 5 and 6 for average latency that achieved by the proposed algorithms.

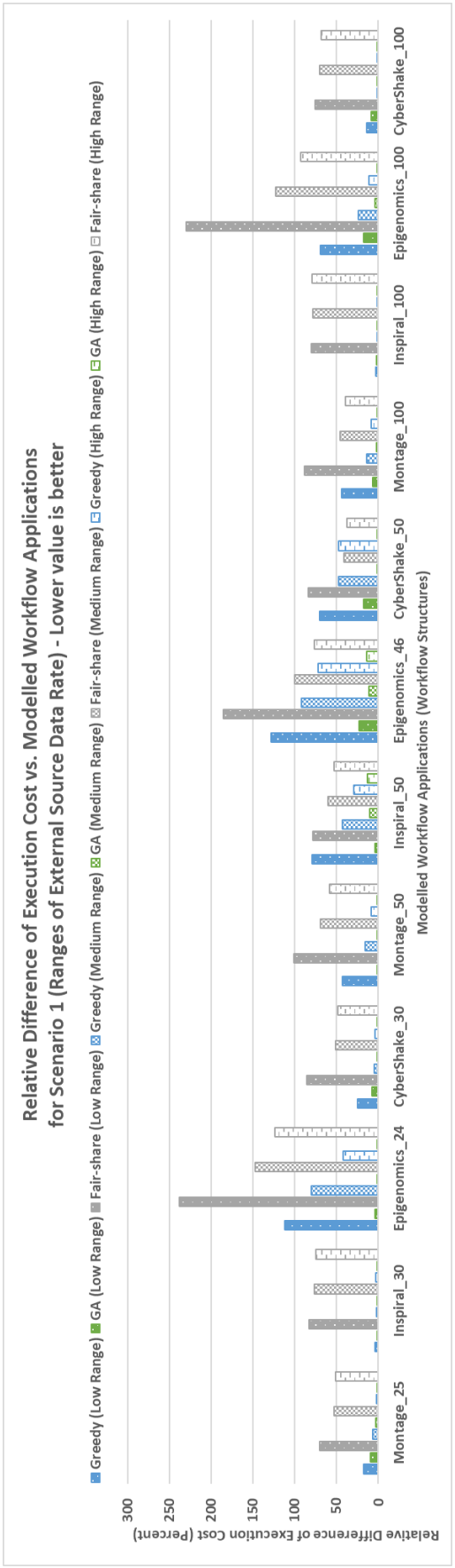


Figure B.1: Execution Cost Comparison for Scenario 1.

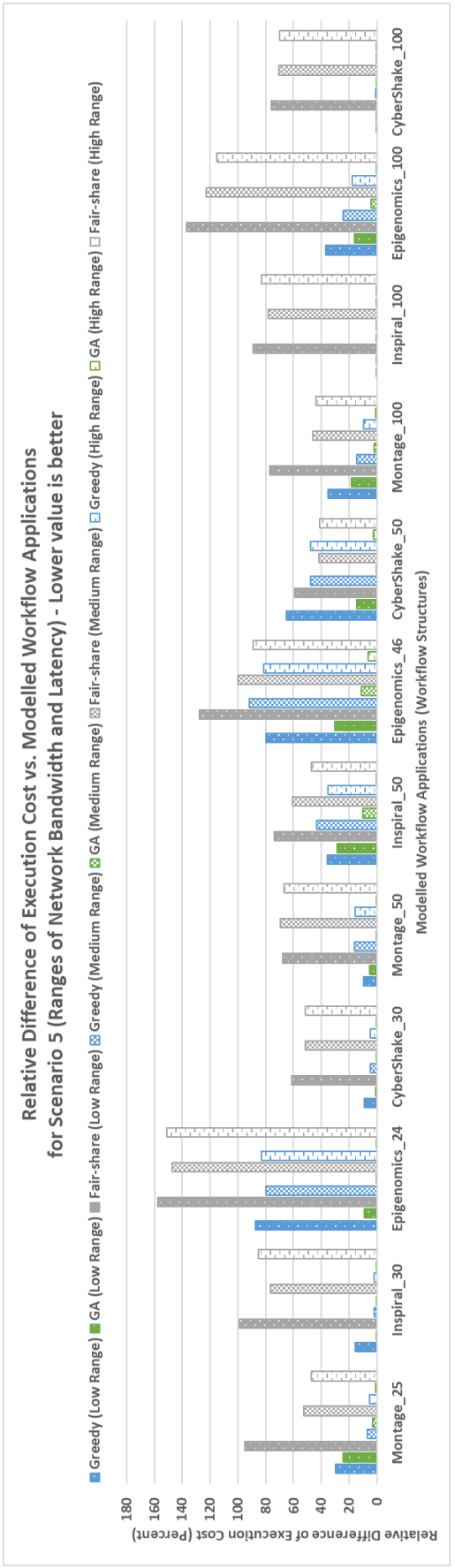


Figure B.2: Execution Cost Comparison for Scenario 5.

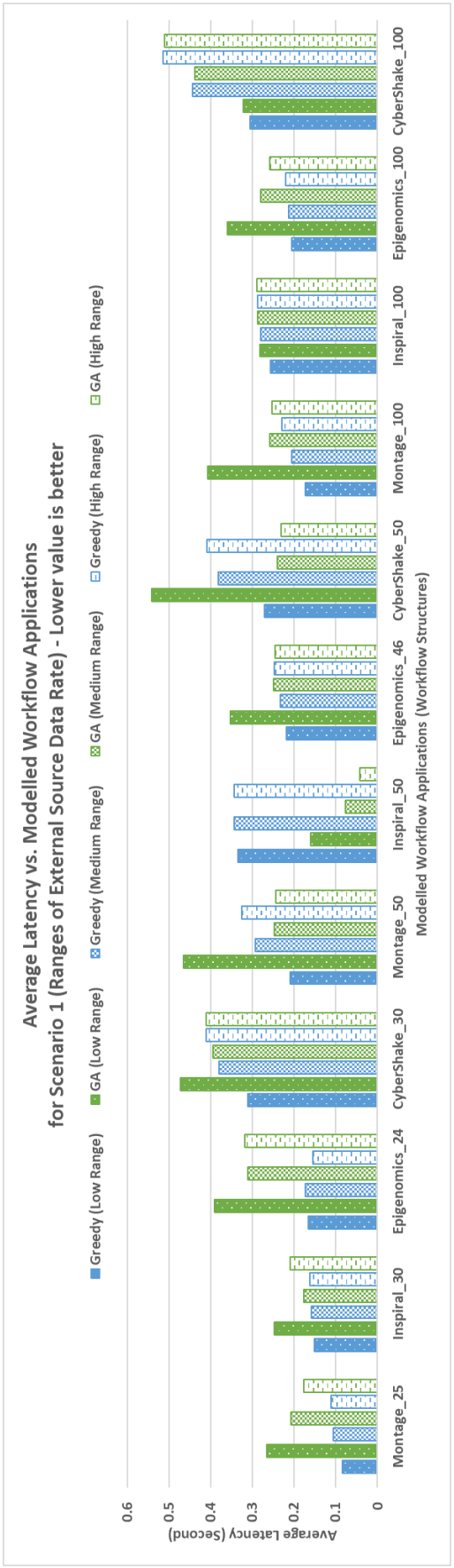


Figure B.3: Proposed algorithms comparison using average end-to-end latency for Scenario 1.

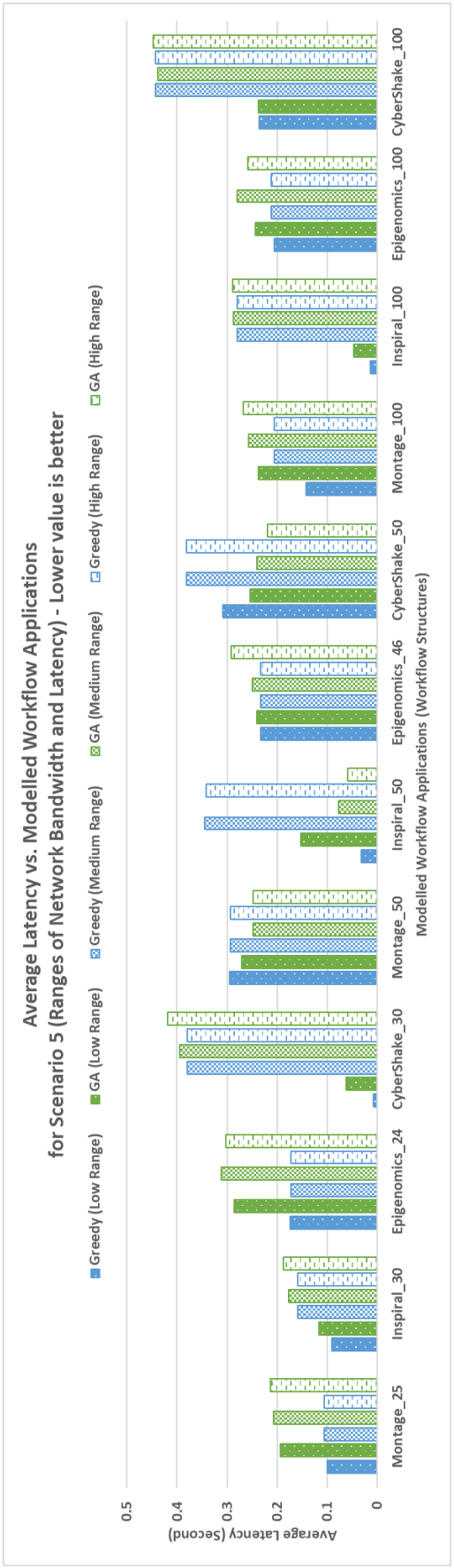


Figure B.4: Proposed algorithms comparison using average end-to-end latency for Scenario 5.

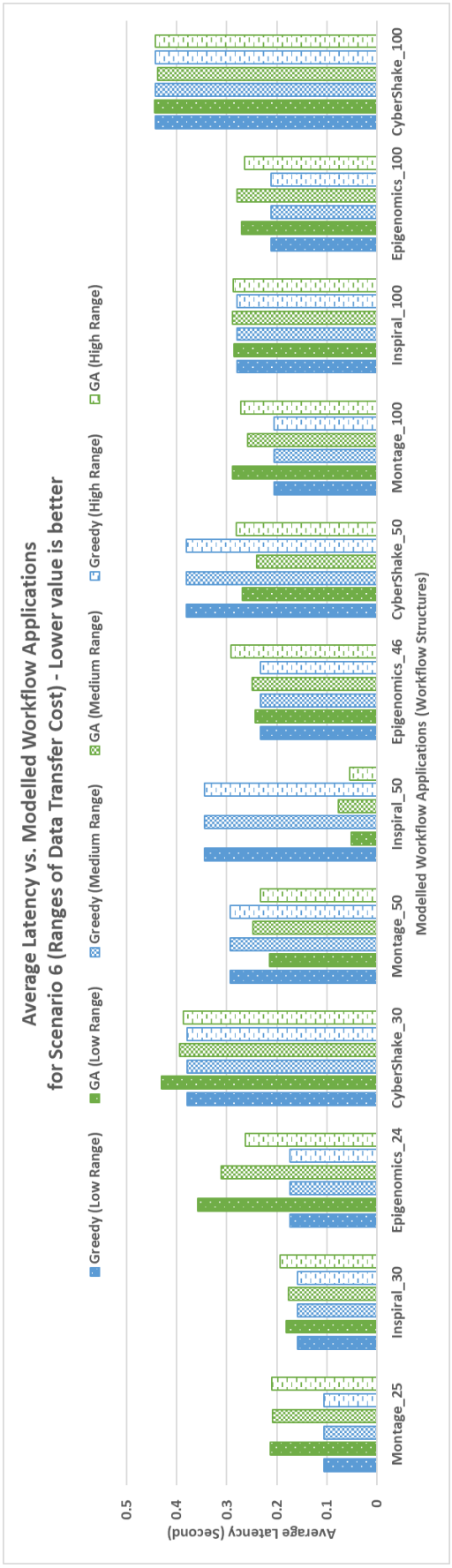


Figure B.5: Proposed algorithms comparison using average end-to-end latency for Scenario 6.

Appendix C

Chapter 5 Appendix

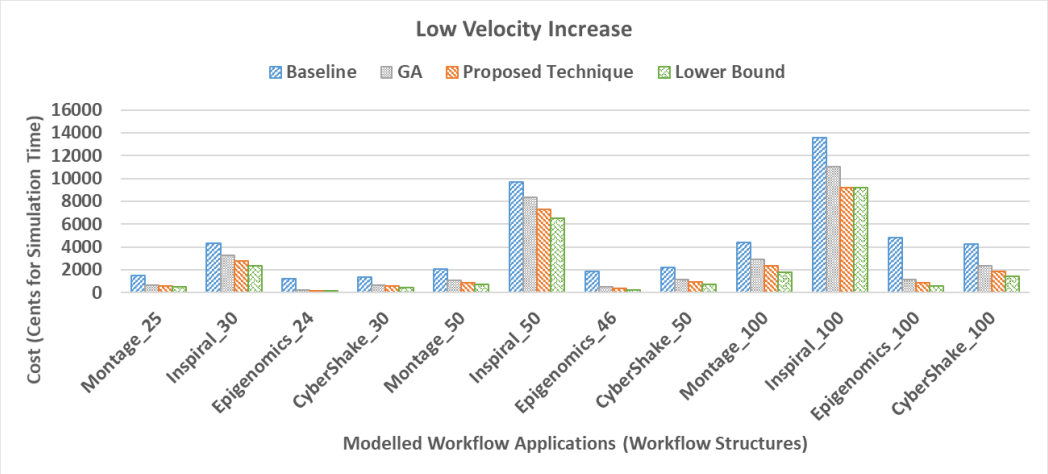


Figure C.1: Total Execution Cost vs. Modelled workflow applications under low range of velocity increase

C.1 More Results for Evaluation 1

Figure C.1 and Figure C.2 show the experimental results for Evaluation 1 with modelled workflow applications under low and high ranges of velocity increase, while Figure C.3 and Figure C.4 show the experimental results for this evolution the modelled workflow applications under low and high ranges of velocity decrease. From these results, it is clear that the proposed technique achieved the lowest total execution cost for all modelled workflow applications under low and high ranges of velocity changes.

C.2 More Results for Evaluation 2

Figure C.5 and Figure C.6 show the experimental results for Evalaution 2 with modelled workflow applications under low and high ranges of velocity increase, while Figure C.7 and Figure C.8 show the experimental results for this evaluation with modelled workflow applications under low and high ranges of velocity decrease. From these results, it is clear that the proposed technique met the process speed required for all modelled workflow applications under low and high ranges of velocity changes.

C.3 More Results for Evaluation 3

Figure C.9 and Figure C.10 show the experimental results for Evaluation 3 with modelled workflow applications under low and high ranges of velocity increase, while Figure C.11 and Figure C.12 show the experimental results for this evaluation with modelled workflow applications under low and high ranges of velocity decrease. From these results, it is clear that the proposed technique is achieved almost the same or near cost reductions per request as achieved by GA. While the proposed technique is unbeatable in term of number of changes required to revise scheduling plan to respond to velocity change request

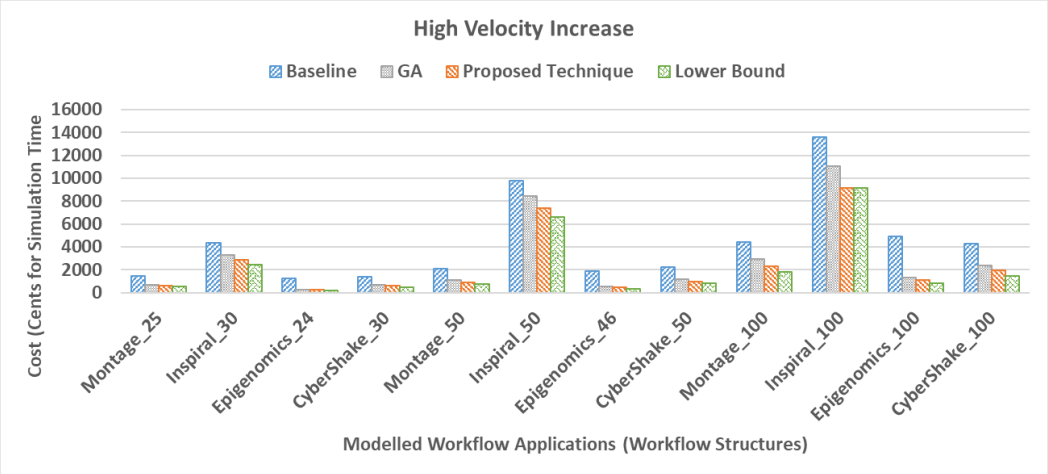


Figure C.2: Total Execution Cost vs. Modelled workflow applications under high range of velocity increase

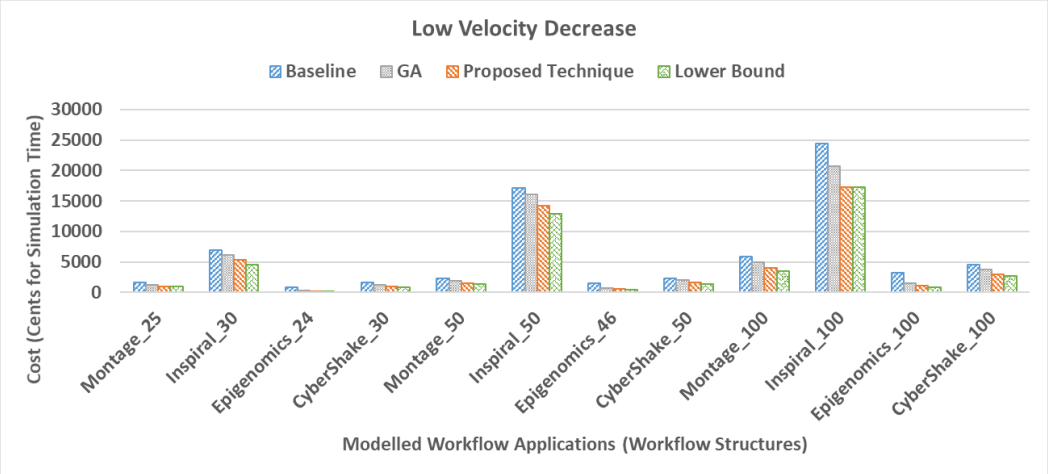


Figure C.3: Total Execution Cost vs. Modelled workflow applications under low range of velocity decrease

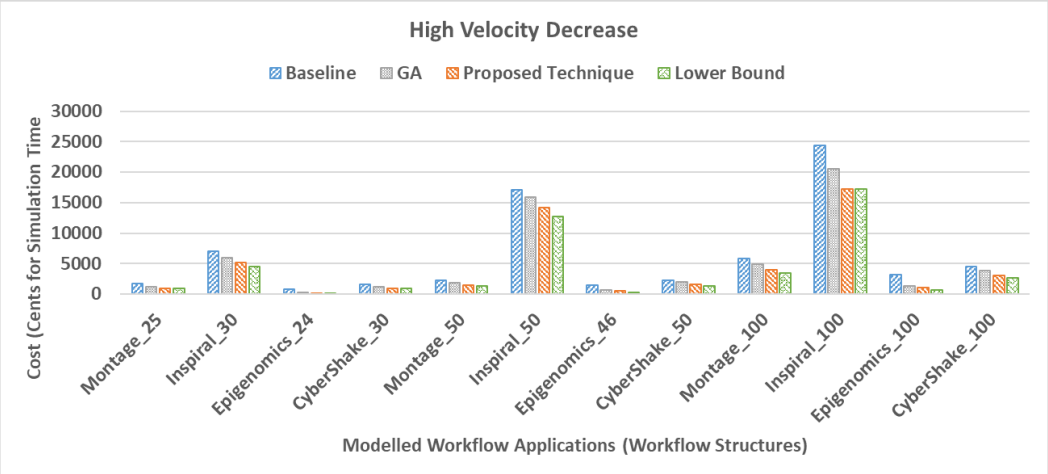


Figure C.4: Total Execution Cost vs. Modelled workflow applications under high range of velocity decrease

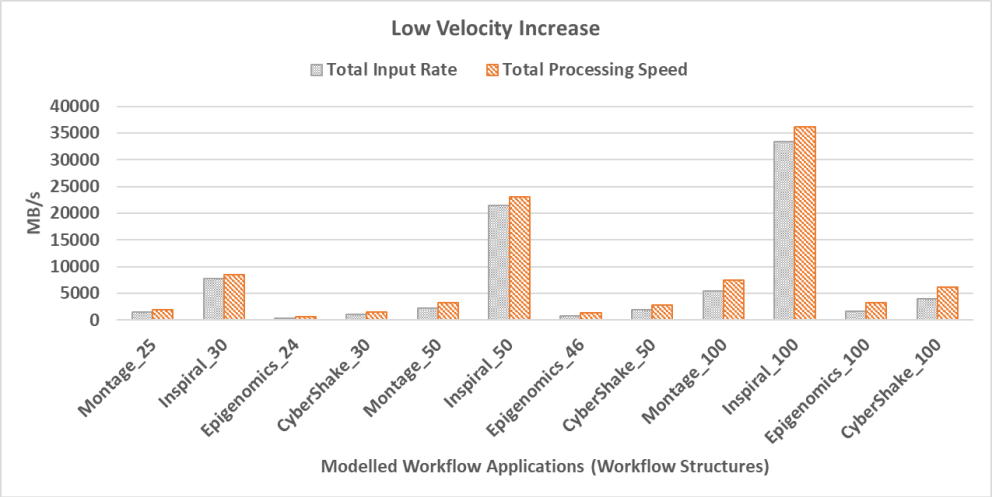


Figure C.5: Total Input Rate vs. Total Processing Speed for different workflow structures (low velocity increase range)

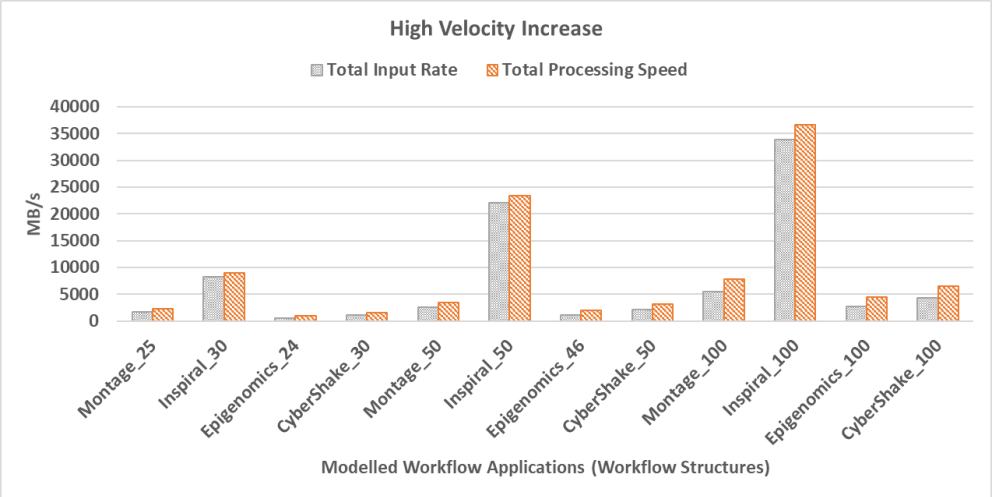


Figure C.6: Total Input Rate vs. Total Processing Speed for different workflow structures (high velocity increase range)

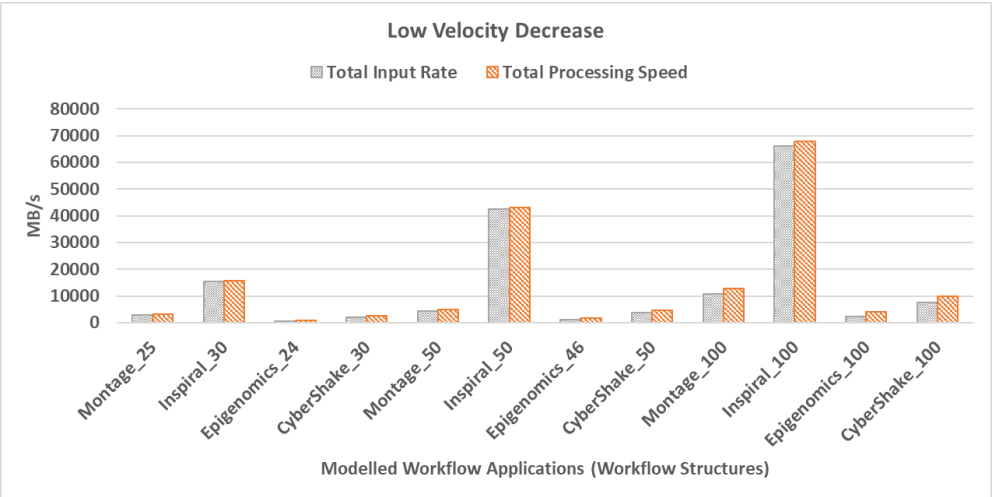


Figure C.7: Total Input Rate vs. Total Processing Speed for different workflow structures (low velocity decrease range)

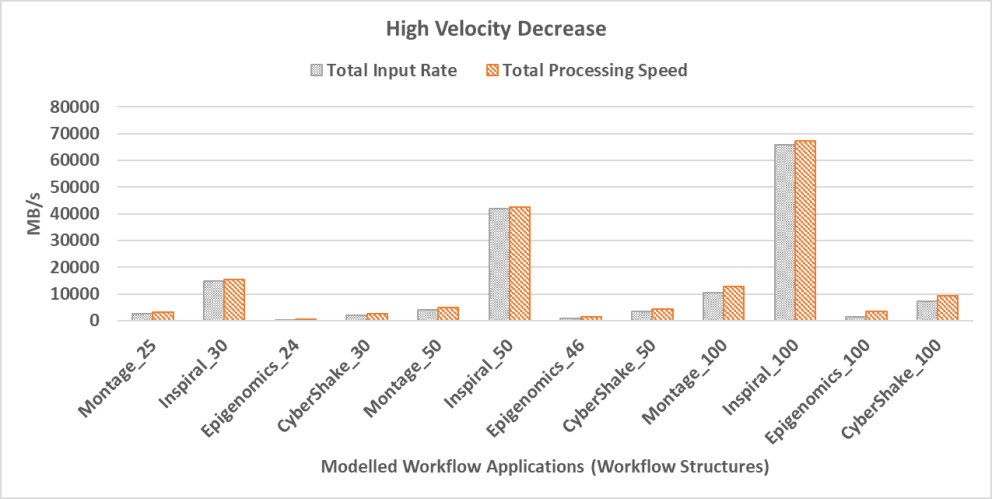
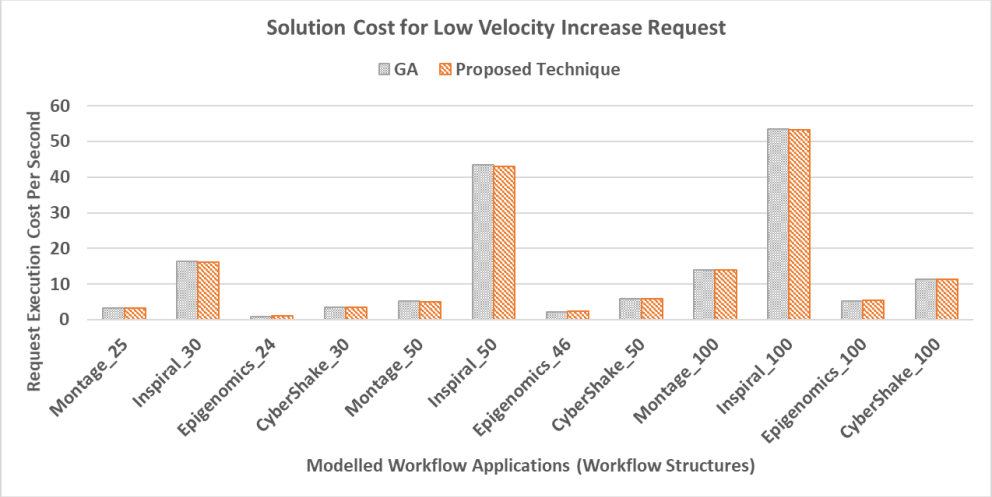
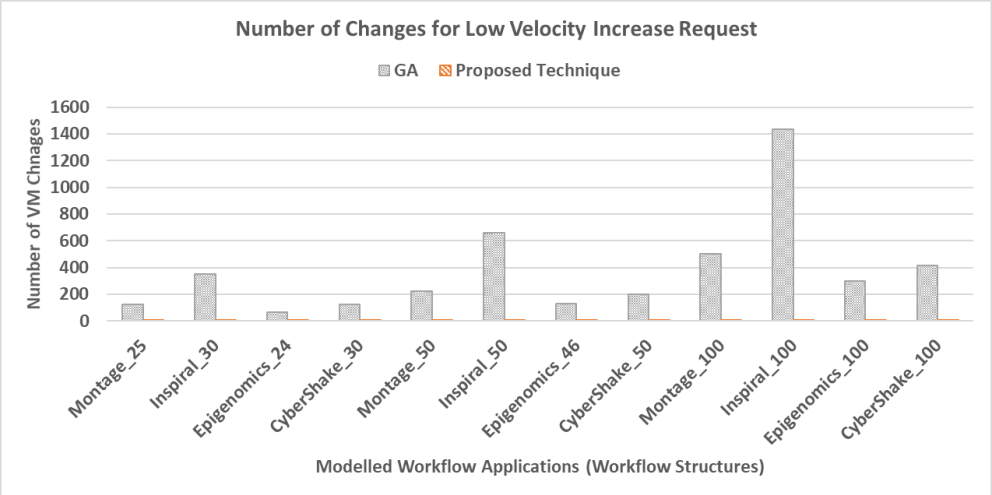


Figure C.8: Total Input Rate vs. Total Processing Speed for different workflow structures (high velocity decrease range)

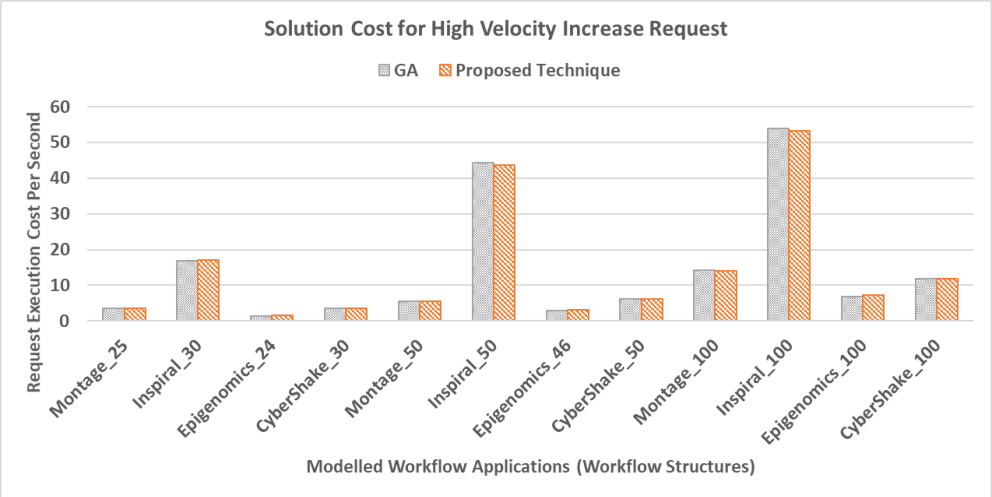


(a) Execution cost

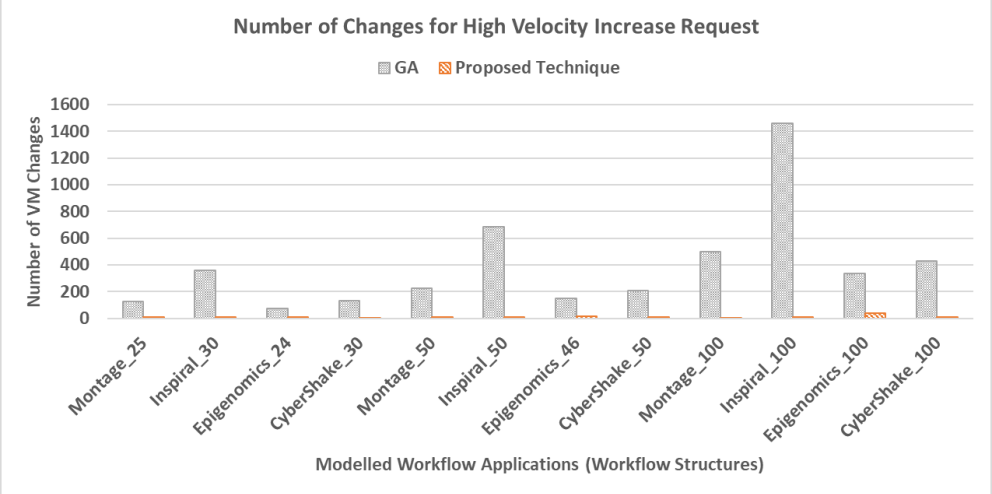


(b) Number of changes

Figure C.9: Quality of solution for different workflow structures (low velocity increase range)

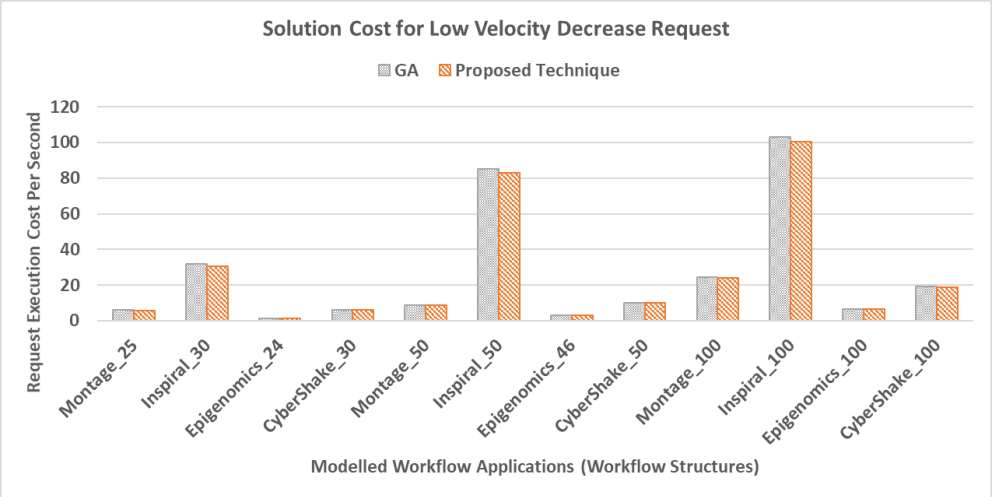


(a) Execution cost

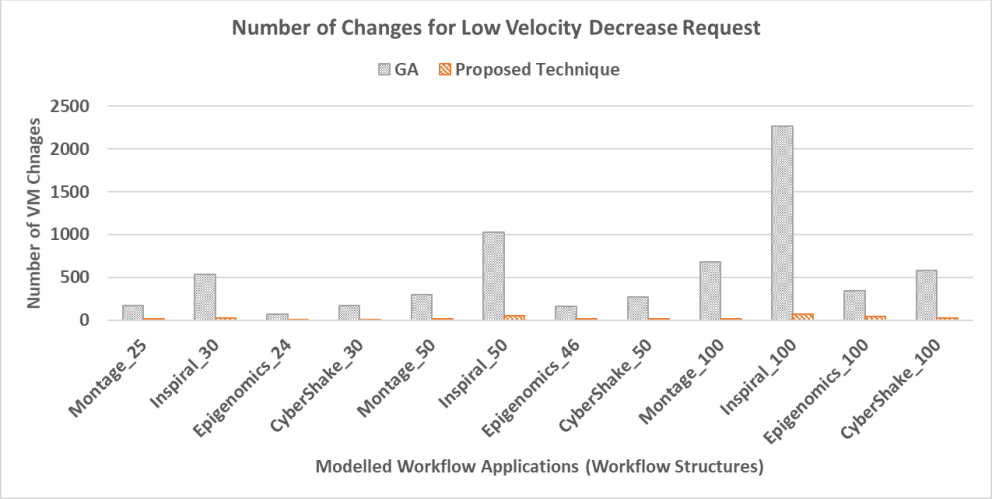


(b) Number of changes

Figure C.10: Quality of solution for different workflow structures (high velocity increase range)

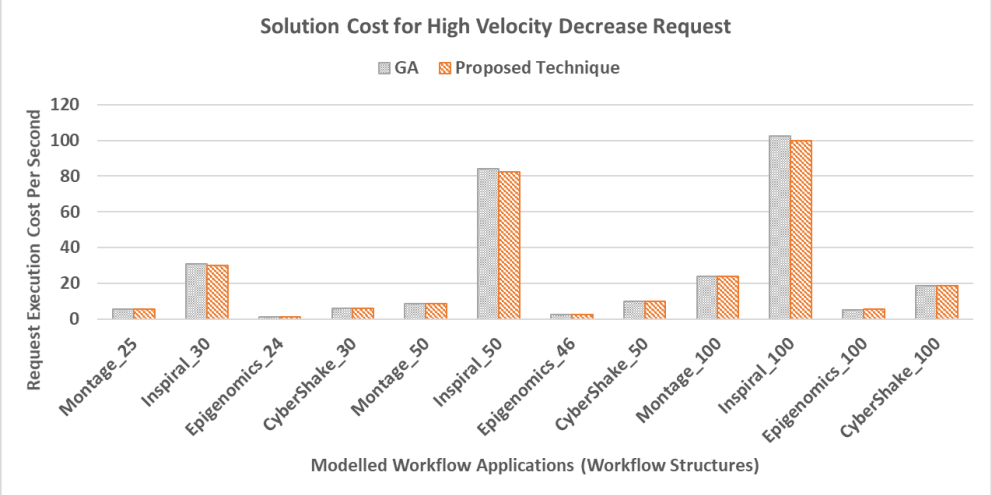


(a) Execution cost

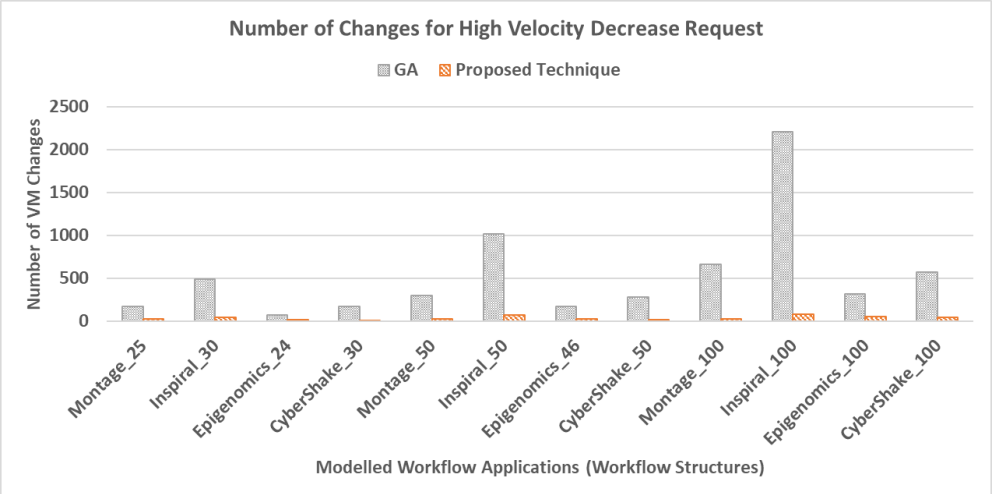


(b) Number of changes

Figure C.11: Quality of solution for different workflow structures (low velocity decrease range)



(a) Execution cost



(b) Number of changes

Figure C.12: Quality of solution for different workflow structures (high velocity decrease range)

Appendix D

Chapter 6 Appendix

D.1 More Quality of Solution Results for Dynamic Form 2 Case 1

Figure D.1 to Figure D.3 show the experimental results for all ranges of service data processing requirement increase for modelled workflow applications, while Figure D.4 to Figure D.6 show those results for all ranges of service data processing requirement decrease.

D.2 More Quality of Solution Results for Dynamic Form 2 Case 2

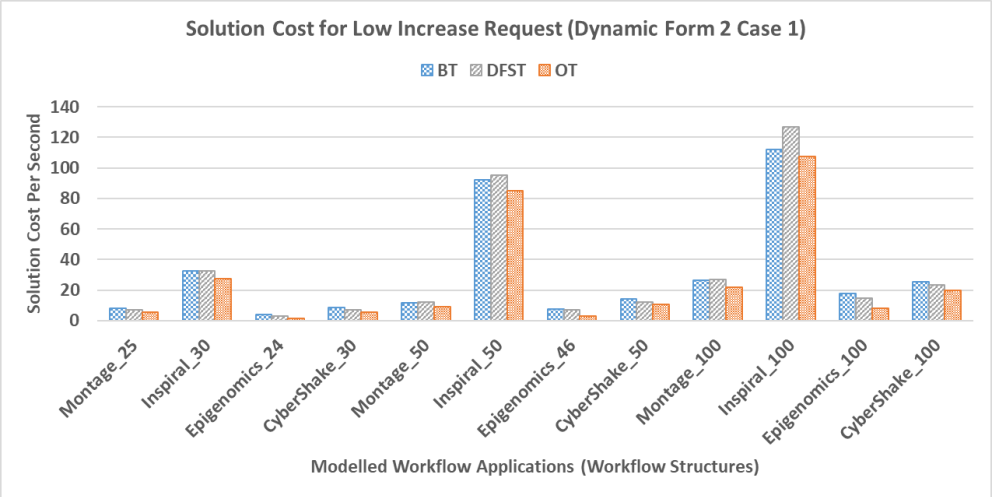
Figure D.7 and Figure D.8 show the experimental results for low and high ranges of service output data rate increase for modelled workflow applications, while Figure D.9 and Figure D.10 show those results for low and high ranges of service output data rate decrease.

D.3 More Quality of Solution Results for Dynamic Form 3 Case 2, 3 and 5

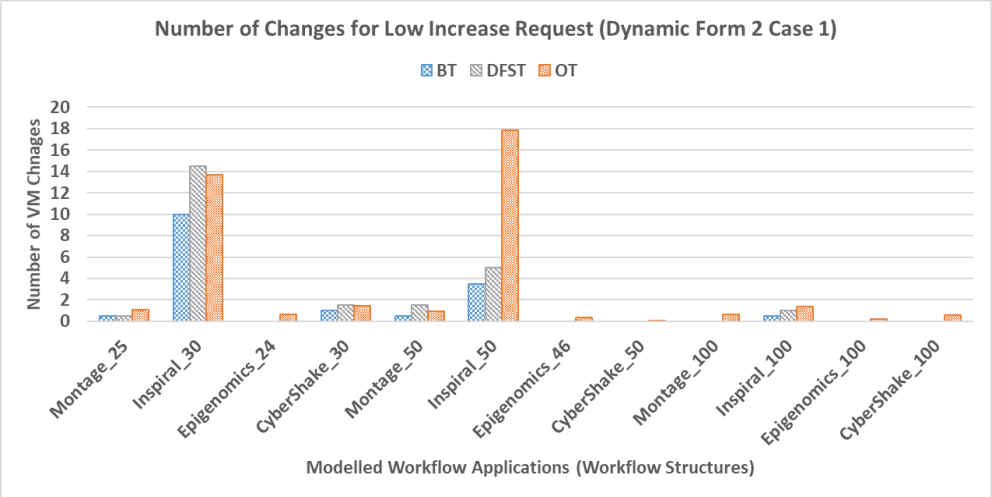
Figure D.11 shows the experimental results for modelled workflow applications under dynamic form 3 case 2., while Figure D.12 shows experimental results for modelled workflow applications under dynamic form 3 case 3. It worth to note that, dynamic form 3 case 3 is not applied on Montage, Inspiral and CyberShake workflows due to their structures (i.e. there is no destination service that meets selection constraint for this case). The quality of solution results for the modelled workflow applications under dynamic form 3 case 5 is presented in Figure D.13.

D.4 More Quality of Solution Results for Dynamic Form 4 Case 2

Figure D.14 shows the experimental results for modelled workflow applications under dynamic form 4 case 2.

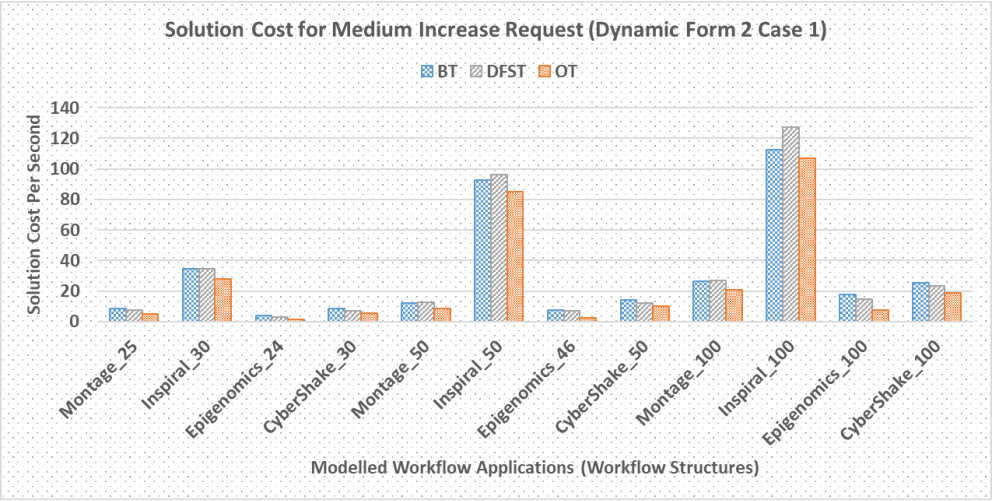


(a) Solution cost

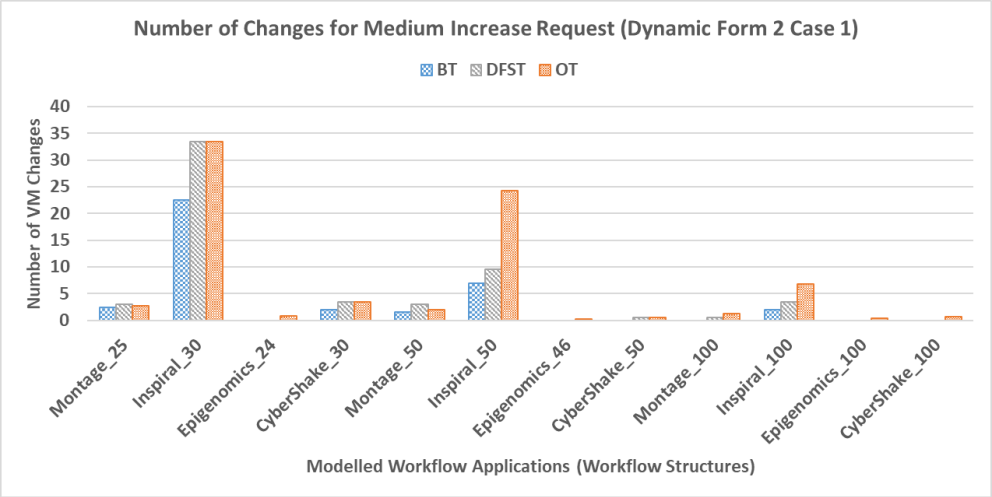


(b) Number of changes

Figure D.1: Quality of solution for different workflow structures under Dynamic Form 2 Case 1 Increase (low range)

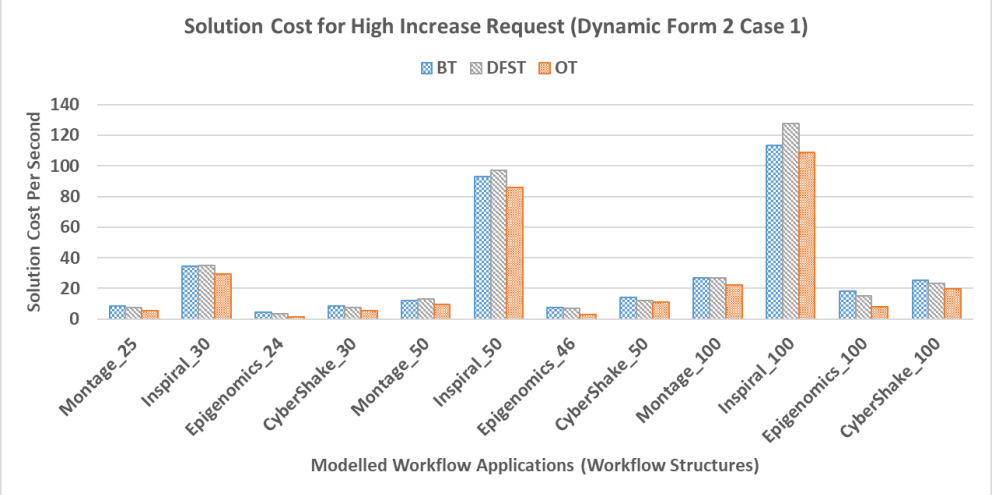


(a) Solution cost

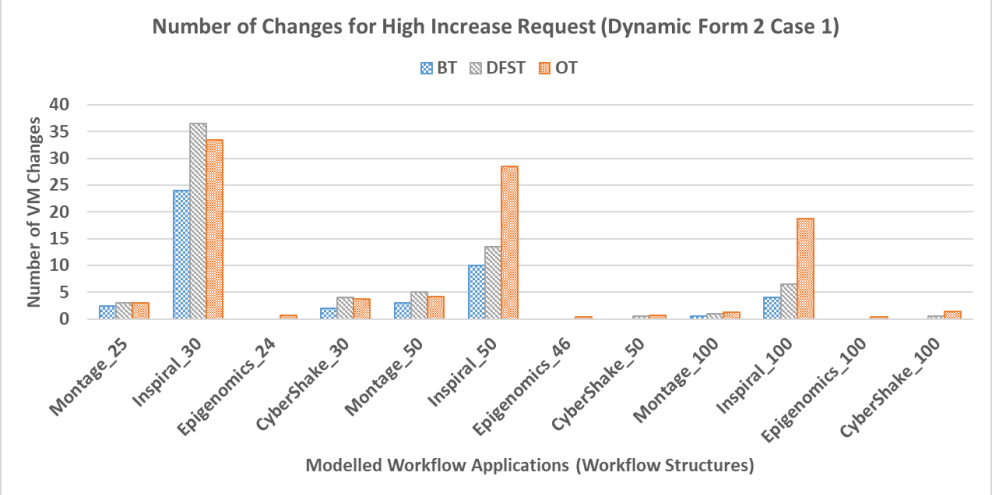


(b) Number of changes

Figure D.2: Quality of solution for different workflow structures under Dynamic Form 2 Case 1 Increase (medium range)

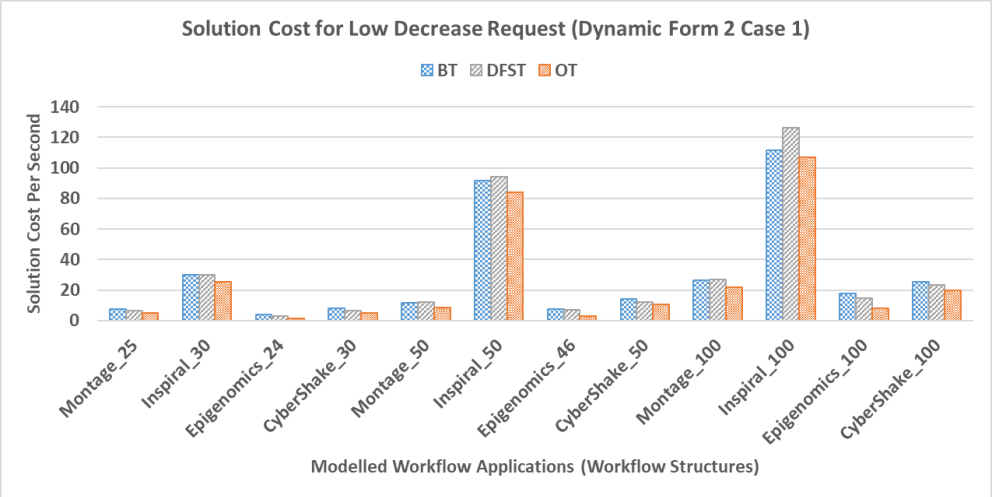


(a) Solution cost

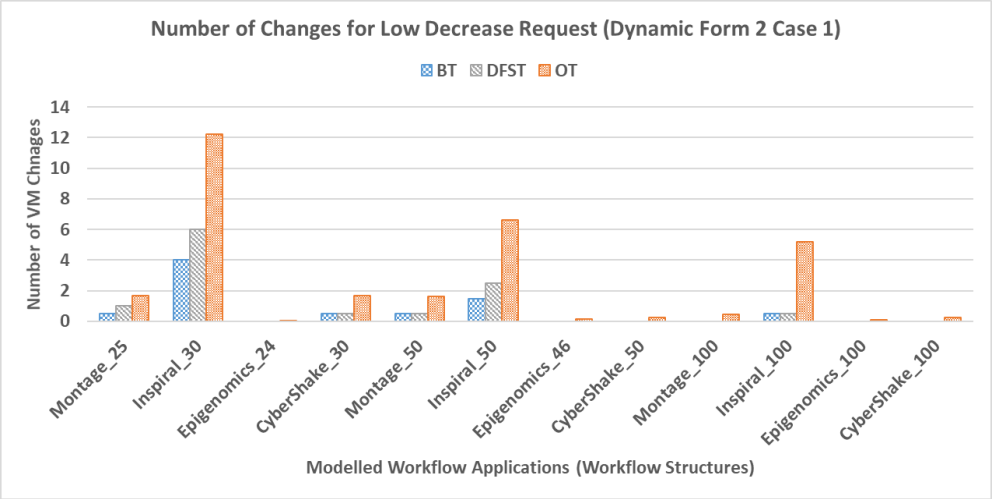


(b) Number of changes

Figure D.3: Quality of solution for different workflow structures under Dynamic Form 2 Case 1 Increase (high range)

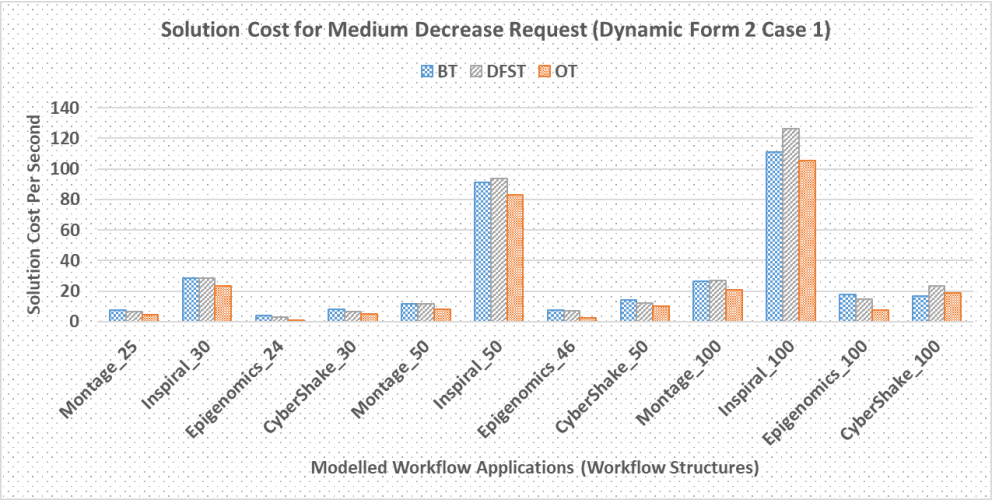


(a) Solution cost

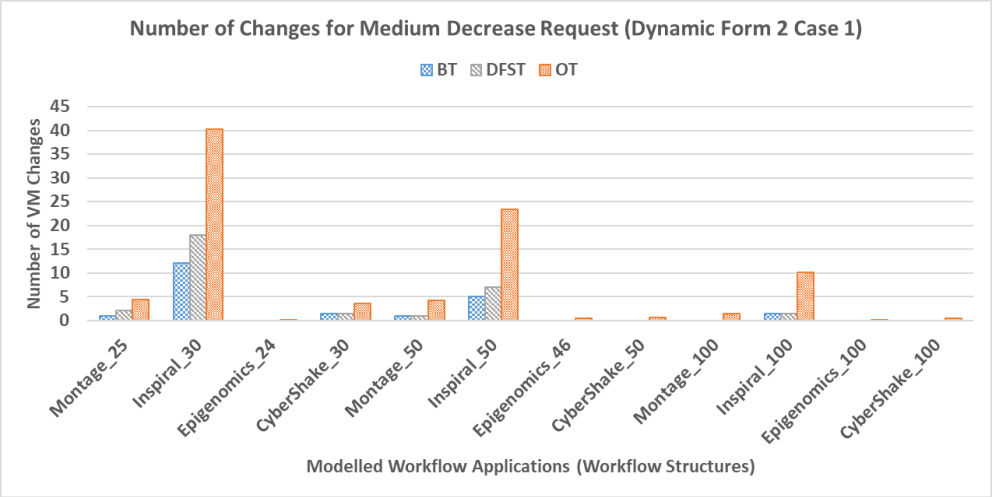


(b) Number of changes

Figure D.4: Quality of solution for different workflow structures under Dynamic Form 2 Case 1 Decrease (low range)

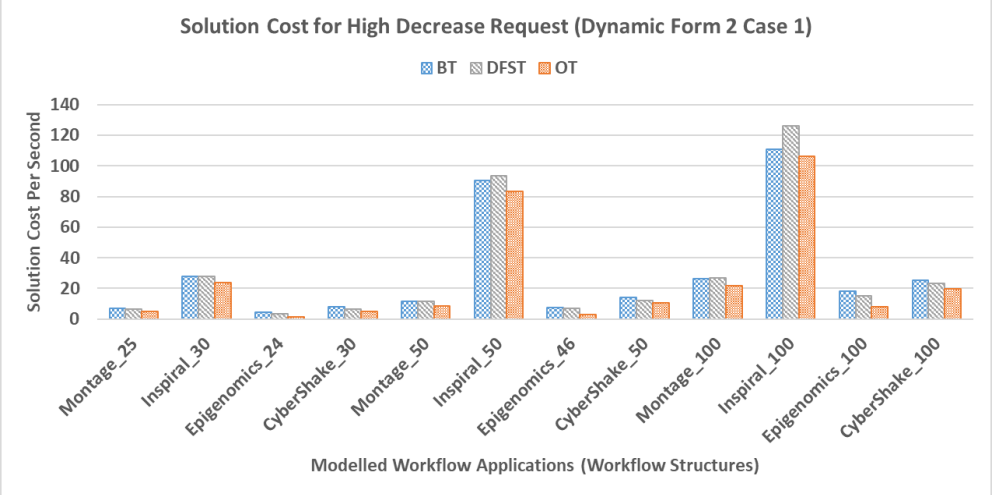


(a) Solution cost

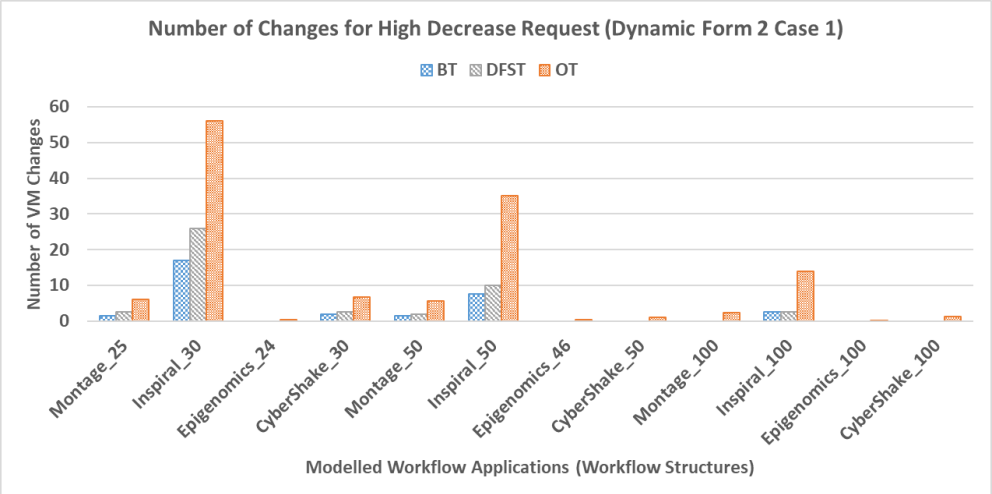


(b) Number of changes

Figure D.5: Quality of solution for different workflow structures under Dynamic Form 2 Case 1 Decrease (medium range)

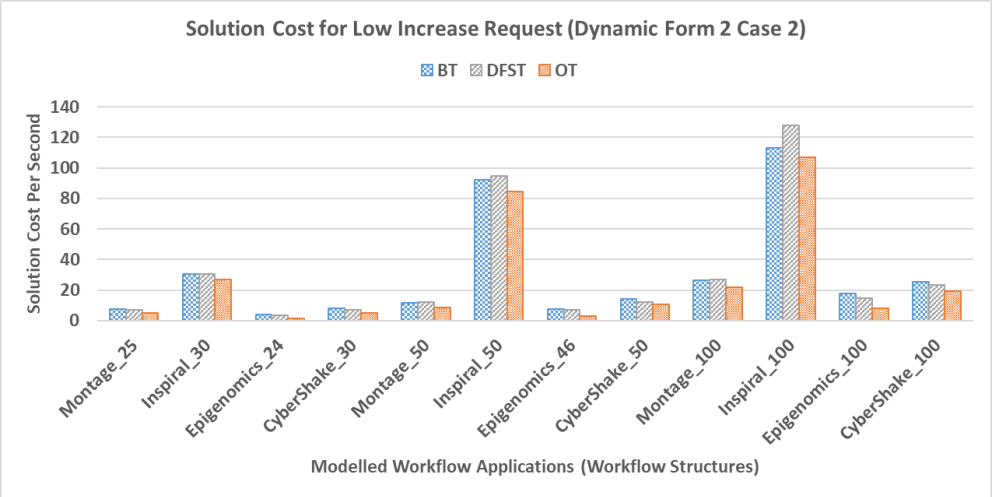


(a) Solution cost

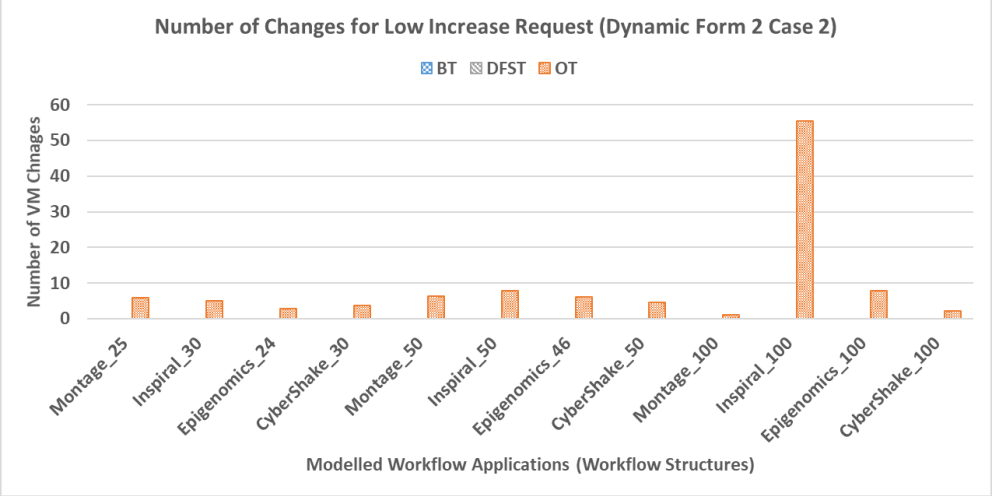


(b) Number of changes

Figure D.6: Quality of solution for different workflow structures under Dynamic Form 2 Case 1 Decrease (high range)

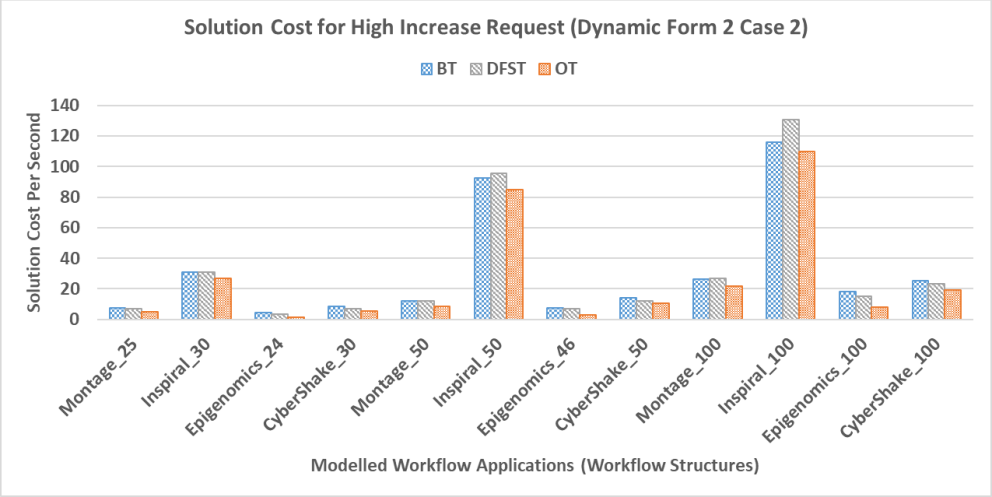


(a) Solution cost

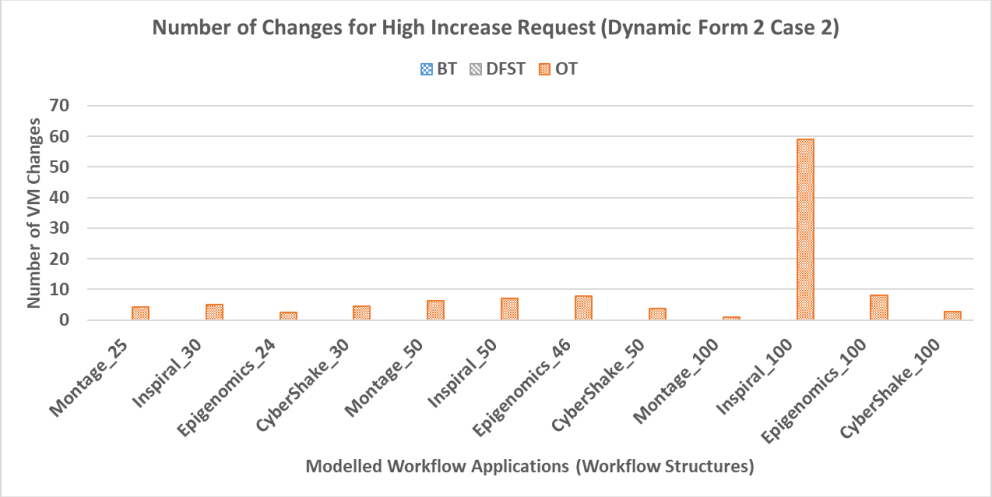


(b) Number of changes

Figure D.7: Quality of solution for different workflow structures under Dynamic Form 2 Case 2 Increase (low range)

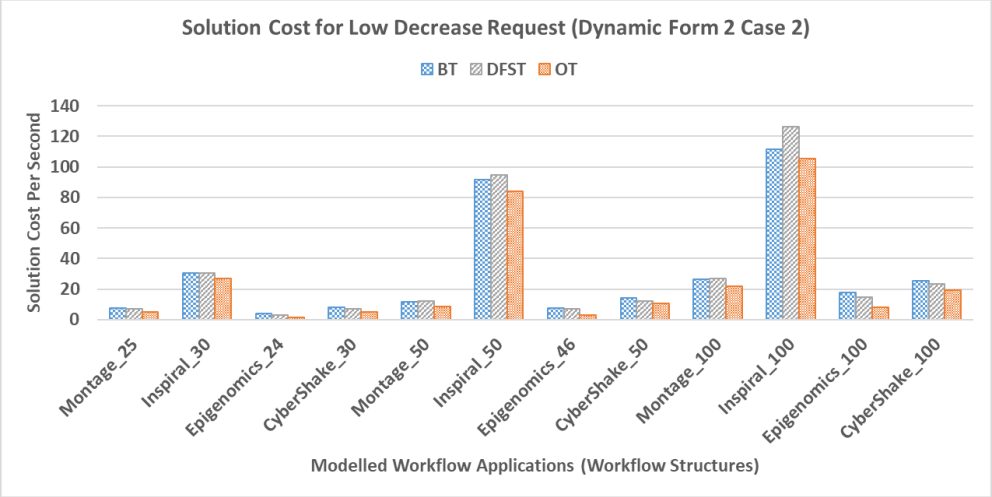


(a) Solution cost

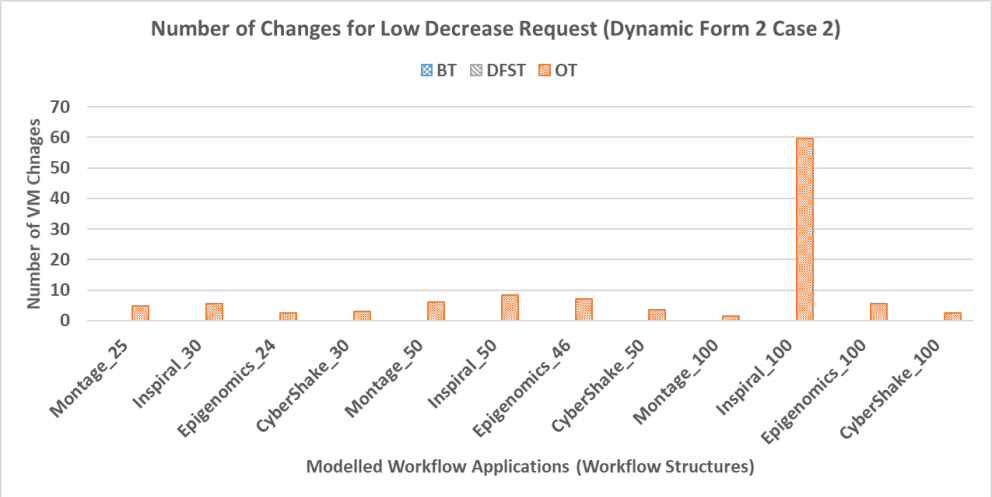


(b) Number of changes

Figure D.8: Quality of solution for different workflow structures under Dynamic Form 2 Case 2 Increase (high range)

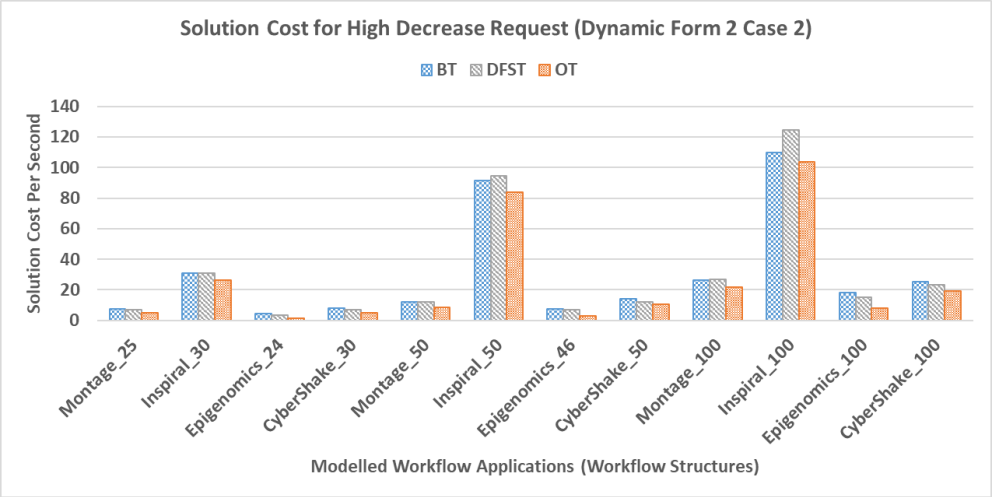


(a) Solution cost

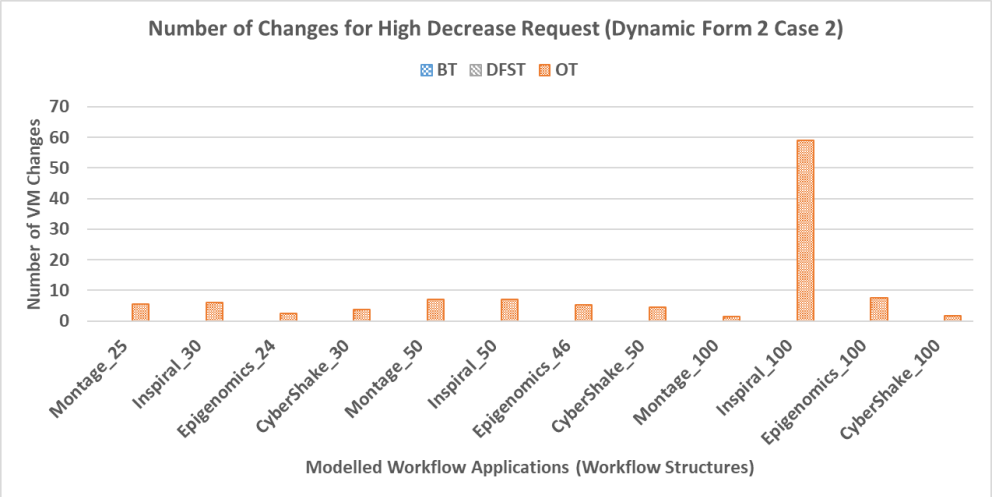


(b) Number of changes

Figure D.9: Quality of solution for different workflow structures under Dynamic Form 2 Case 2 Decrease (low range)

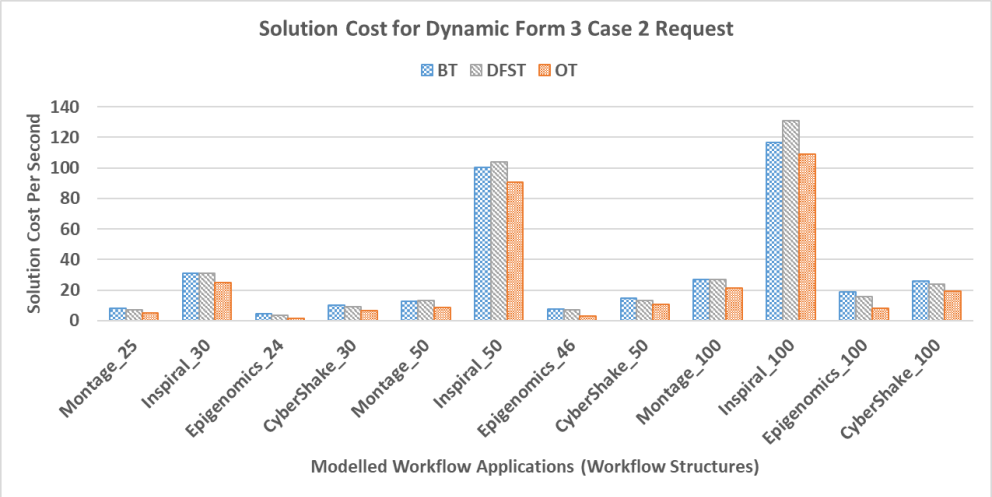


(a) Solution cost

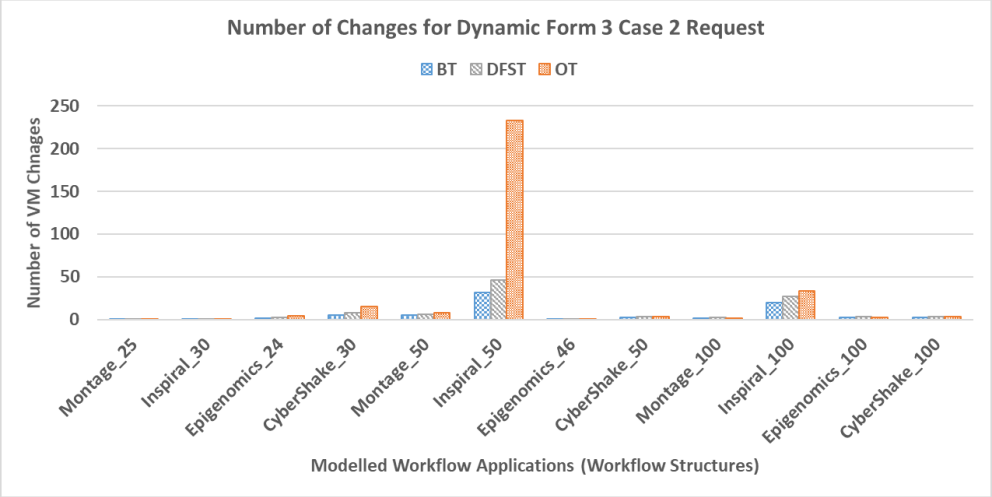


(b) Number of changes

Figure D.10: Quality of solution for different workflow structures under Dynamic Form 2 Case 2 Decrease (high range)

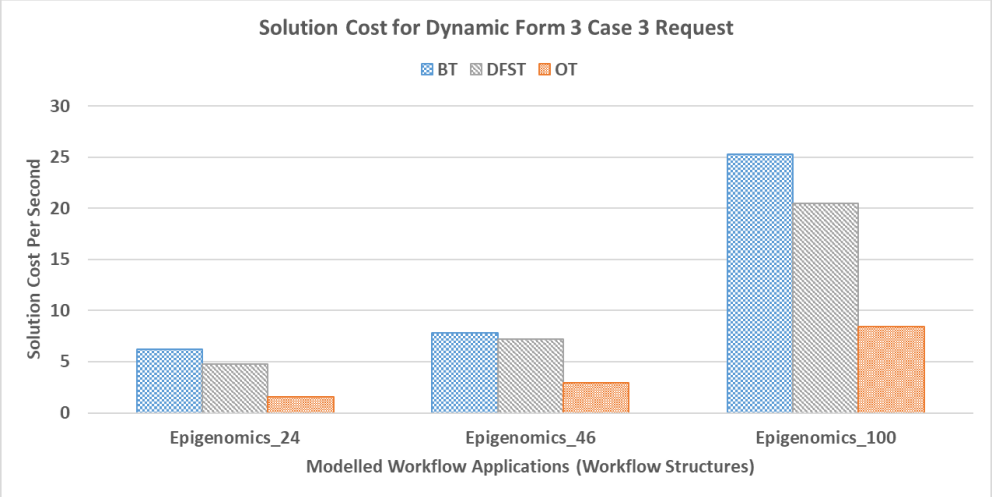


(a) Solution cost

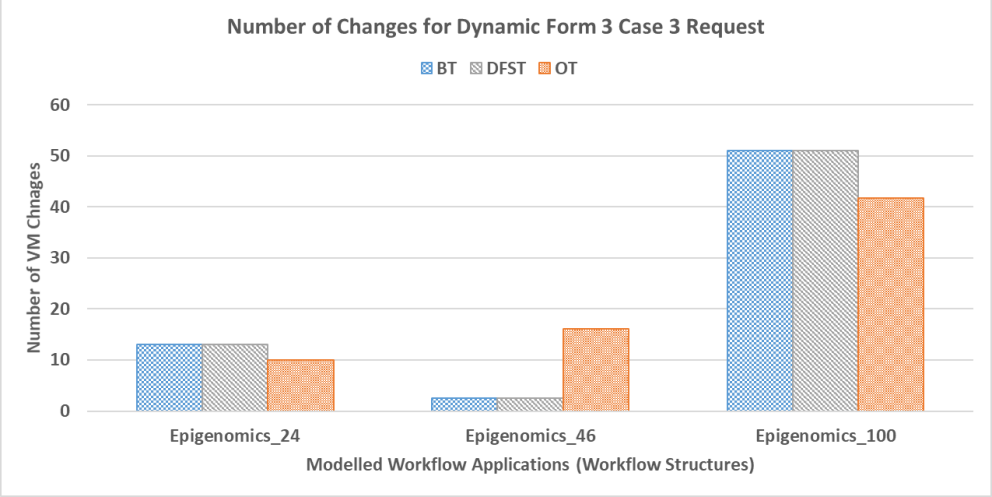


(b) Number of changes

Figure D.11: Quality of solution for different workflow structures under Dynamic Form 3 Case 2

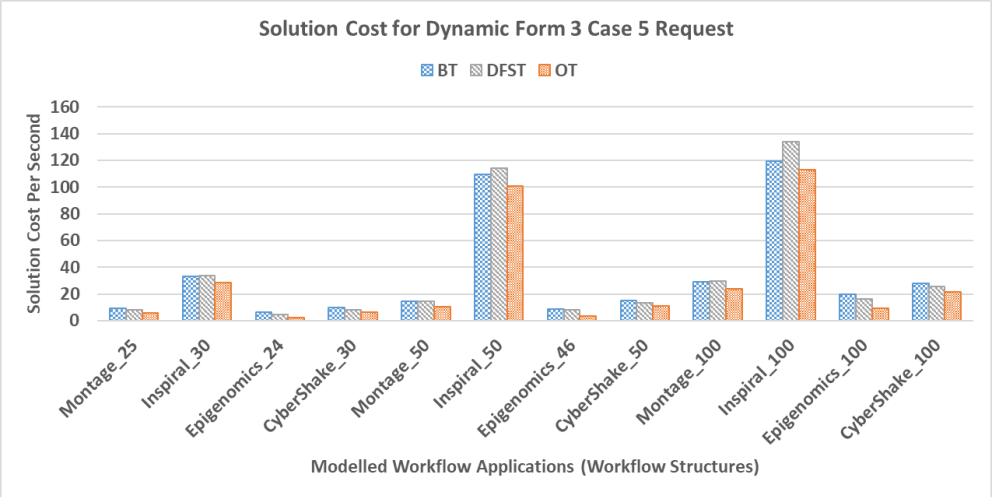


(a) Solution cost

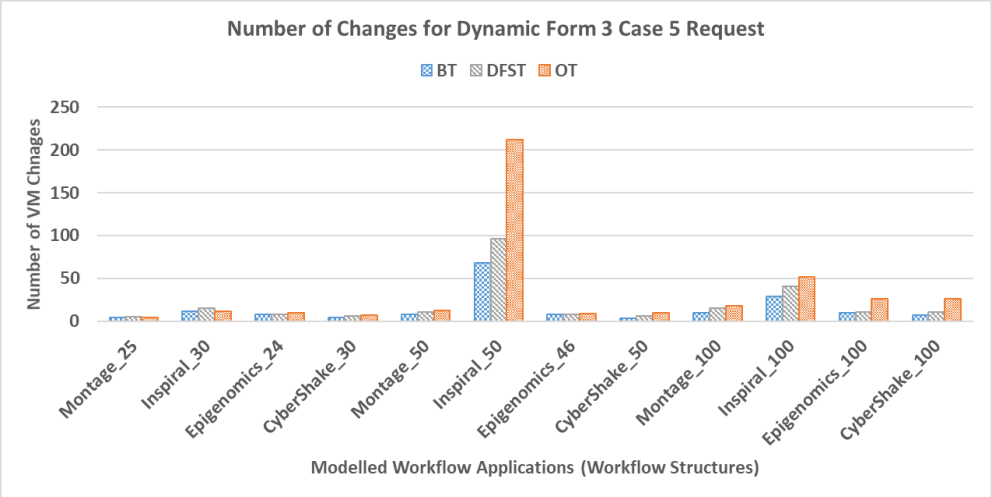


(b) Number of changes

Figure D.12: Quality of solution for different workflow structures under Dynamic Form 3 Case 3

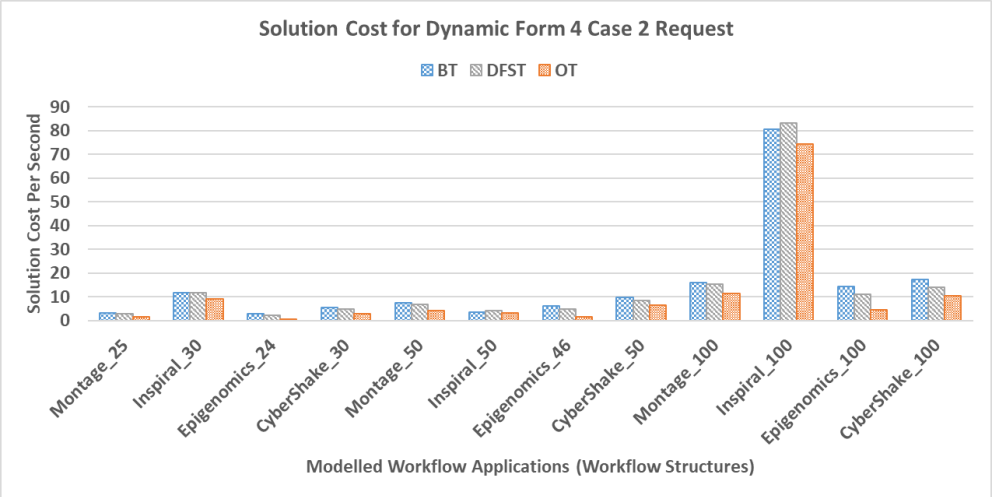


(a) Solution cost

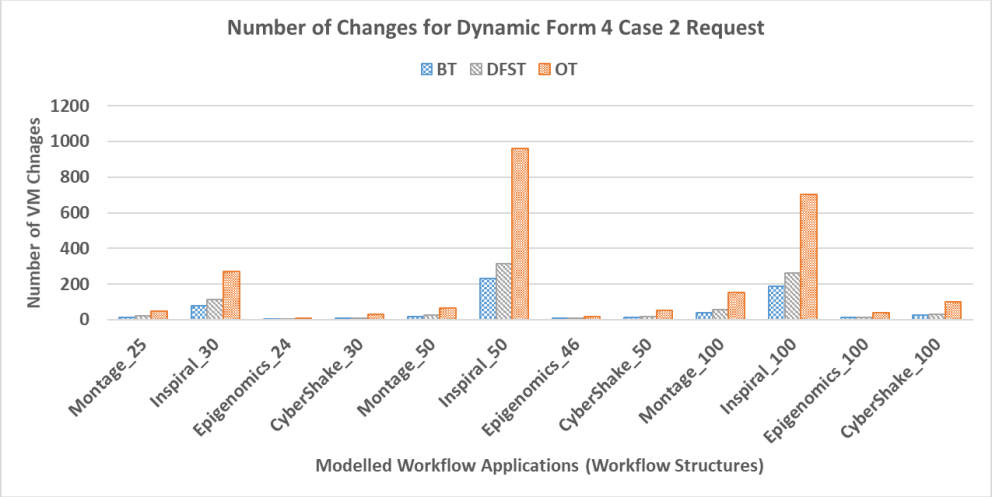


(b) Number of changes

Figure D.13: Quality of solution for different workflow structures under Dynamic Form 3 Case 5



(a) Solution cost



(b) Number of changes

Figure D.14: Quality of solution for different workflow structures under Dynamic Form 4 Case 2

Bibliography

(2015). Anomaly detection over sensor data streams. `http://wiki.clommunity-project.eu/pilots:and`.

(2017). Nectar cloud. `https://nectar.org.au/`.

Adamu, F. B., Habbal, A., Hassan, S., Malaysia, U. U., Cottrell, R. L., White, B., Abdullah, I., Malaysia, U. U., et al. (2016). A survey on big data indexing strategies. Technical report, SLAC National Accelerator Lab., Menlo Park, CA (United States).

Ahmad, S. G., Liew, C. S., Rafique, M. M., and Munir, E. U. (2017). Optimization of data-intensive workflows in stream-based data processing models. *The Journal of Supercomputing*, 73(9):3901–3923.

Ahmad, S. G., Liew, C. S., Rafique, M. M., Munir, E. U., and Khan, S. U. (2014). Data-intensive workflow optimization based on application task graph partitioning in heterogeneous computing systems. In *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*, pages 129–136. IEEE.

Ahuja, S. P. and Kaza, B. (2015). Performance evaluation of data intensive computing in the cloud. In *Cloud Technology: Concepts, Methodologies, Tools, and Applications*, pages 1901–1914. IGI Global.

Akin, D., Sisiopiku, V. P., and Skabardonis, A. (2011). Impacts of weather on traffic flow characteristics of urban freeways in istanbul. *Procedia-Social and Behavioral Sciences*, 16:89–99.

Albertsson, L. (2016). Data pipelines from zero to solid.

Albrecht, M., Donnelly, P., Bui, P., and Thain, D. (2012). Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In *1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, page 1. ACM.

Alrokayan, M., Dastjerdi, A. V., and Buyya, R. (2014). Sla-aware provisioning and scheduling of cloud resources for big data analytics. In *2014 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 1–8. IEEE.

Amazon (2017a). Amazon ec2 pricing. `https://aws.amazon.com/ec2/pricing/`.

Amazon (2017b). Aws lambda. `https://aws.amazon.com/lambda/details/`.

Amini, S., Gerostathopoulos, I., and Prehofer, C. (2017). Big data analytics architecture for real-time traffic control. In *2017 5th IEEE International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS)*, pages 710–715. IEEE.

Amstutz, P., Andeer, R., Chapman, B., Chilton, J., Crusoe, M. R., Valls Guimera, R., Carrasco Hernandez, G., Ivkovic, S., Kartashov, A., Kern, J., et al. (2016). Common workflow language, draft 3.

Apache (2017). Apache hadoop yarn. `https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/YARN.html`.

Barika, M., Garg, S., Chan, A., and Calheiros, R. (2019a). Scheduling algorithms for efficient execution of stream workflow applications in multicloud environments. *IEEE Transactions on Services Computing*.

Barika, M., Garg, S., Chan, A., Calheiros, R. N., and Ranjan, R. (2019b). Iotsim-stream: Modelling stream graph application in cloud simulation. *Future Generation Computer Systems*, 99:86–105.

Barika, M., Garg, S., Zomaya, A. Y., Wang, L., van Moorsel, A., and Ranjan, R. (2019c). Orchestrating big data analysis workflows in the cloud: research challenges, survey, and future directions. *ACM Computing Surveys*, pages 1–37.

Barker, A. and Van Hemert, J. (2007). Scientific workflow: a survey and research directions. In *International Conference on Parallel Processing and Applied Mathematics*, pages 746–753. Springer.

Beloglazov, A., Abawajy, J., and Buyya, R. (2012). Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future generation computer systems*, 28(5):755–768.

Berthold, H., Schmidt, S., Lehner, W., and Hamann, C.-J. (2005). Integrated resource management for data stream systems. In *2005 ACM symposium on Applied computing*, pages 555–562. ACM.

Bessani, A., Correia, M., Quaresma, B., André, F., and Sousa, P. (2013). Depsky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)*, 9(4):12.

Bessani, A., Mendes, R., Oliveira, T., Neves, N., Correia, M., Pasin, M., and Verissimo, P. (2014). Scfs: A shared cloud-backed file system. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 169–180.

Bhagwanani, S. (2005). An evaluation of end-user interfaces of scientific workflow management systems. *Master thesis, North Carolina State University.*

Bharathi, S., Chervenak, A., Deelman, E., Mehta, G., Su, M.-H., and Vahi, K. (2008). Characterization of scientific workflows. In *2008 third workshop on workflows in support of large-scale science*, pages 1–10. IEEE.

Bhuvaneshwar, K., Sulakhe, D., Gauba, R., Rodriguez, A., Madduri, R., Dave, U., Lacinski, L., Foster, I., Gusev, Y., and Madhavan, S. (2015). A case study for cloud based high throughput analysis of ngs data using the globus genomics system. *Computational and structural biotechnology journal*, 13:64–74.

Bicer, T., Yin, J., Chiu, D., Agrawal, G., and Schuchardt, K. (2013). Integrating online compression to accelerate large-scale data analytics applications. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1205–1216. IEEE.

Bohli, J.-M., Gruschka, N., Jensen, M., Iacono, L. L., and Marnau, N. (2013). Security and privacy-enhancing multicloud architectures. *IEEE Transactions on Dependable and Secure Computing*, 10(4):212–224.

Bożek, A. and Werner, F. (2018). Flexible job shop scheduling with lot streaming and sublot size optimisation. *International Journal of Production Research*, 56(19):6391–6411.

Buddhika, T., Stern, R., Lindburg, K., Ericson, K., and Pallickara, S. (2017). Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3553–3569.

Bux, M. and Leser, U. (2013). Parallelization in scientific workflow management systems. *arXiv preprint arXiv:1303.7195.*

Cafaro, M. and Aloisio, G. (2011). Grids, clouds, and virtualization. In *Grids, Clouds and Virtualization*, pages 1–21. Springer.

Cai, H., Xu, B., Jiang, L., and Vasilakos, A. V. (2017). Iot-based big data storage systems in cloud computing: perspectives and challenges. *IEEE Internet of Things Journal*, 4(1):75–87.

Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A., and Buyya, R. (2011). Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50.

Cao, Y. Y. J., Venugopal, S., Benatallah, B., and Chen, J. (2016). A resource provisioning strategy for elastic analytical workflows in the cloud. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 538–545. IEEE.

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4).

Cardellini, V., Grassi, V., Lo Presti, F., and Nardelli, M. (2016). Optimal operator placement for distributed stream processing applications. In *10th ACM International Conference on Distributed and Event-based Systems*, pages 69–80. ACM.

Casanova, H., Pandey, S., Oeth, J., Tanaka, R., Suter, F., and da Silva, R. F. (2018). Wrench: A framework for simulating workflow management systems. In *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, pages 74–85. IEEE.

Chang, H.-C., Li, K.-C., Lin, Y.-L., Yang, C.-T., Wang, H.-H., and Lee, L.-T. (2005). Performance issues of grid computing based on different architecture cluster computing platforms. In *19th International Conference on Advanced Information Networking and Applications (AINA'05) Volume 1 (AINA papers)*, volume 2, pages 321–324. IEEE.

Chen, C., Luan, T. H., Guan, X., Lu, N., and Liu, Y. (2017). Connected vehicular transportation: Data analytics and traffic-dependent networking. *IEEE Vehicular Technology Magazine*, 12(3):42–54.

Chen, H., Wen, J., Pedrycz, W., and Wu, G. (2018a). Big data processing workflows oriented real-time scheduling algorithm using task-duplication in geo-distributed clouds. *IEEE Transactions on Big Data*.

Chen, J., Chen, Y., Du, X., Li, C., Lu, J., Zhao, S., and Zhou, X. (2013). Big data challenge: a data management perspective. *Frontiers of Computer Science*, 7(2):157–164.

Chen, L., Liu, S., Li, B., and Li, B. (2018b). Scheduling jobs across geo-distributed datacenters with max-min fairness. *IEEE Transactions on Network Science and Engineering*.

Chen, P. (2016). Big data analytics in static and streaming provenance.

Chen, W. and Deelman, E. (2011). Partitioning and scheduling workflows across multiple sites with storage constraints. In *International Conference on Parallel Processing and Applied Mathematics*, pages 11–20. Springer.

Chen, W. and Deelman, E. (2012a). Integration of workflow partitioning and resource provisioning. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 764–768. IEEE.

Chen, W. and Deelman, E. (2012b). Workflowsim: A toolkit for simulating scientific workflows in distributed environments. In *2012 IEEE 8th International Conference on E-Science*, pages 1–8. IEEE.

Cofer, H., Caccamo, M., Fretter, P., and Stitt, T. (2015). The impact of large shared memory computing architectures in genomics workflows. White paper, Silicon Graphics International (SGI).

Collins, K. (2015). When to use containers or virtual machines, and why.

Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K., and Sears, R. (2010). Mapreduce online. In *Nsdi*, volume 10, page 20.

Convolbo, M. W., Chou, J., Hsu, C.-H., and Chung, Y. C. (2018). Geodis: towards the optimization of data locality-aware job scheduling in geo-distributed data centers. *Computing*, 100(1):21–46.

Costa, F., de Oliveira, D., and Mattoso, M. (2014). Towards an adaptive and distributed architecture for managing workflow provenance data. In *2014 IEEE 10th International Conference on e-Science*, volume 2, pages 79–82. IEEE.

Costa, P., Pasin, M., Bessani, A. N., and Correia, M. (2011). Byzantine fault-tolerant mapreduce: Faults are not just crashes. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 32–39. IEEE.

Curcin, V. and Ghanem, M. (2008). Scientific workflow systems-can one size fit all? In *2008 Cairo International Biomedical Engineering Conference*, pages 1–9. IEEE.

Cuzzocrea, A. (2014). Privacy and security of big data: current challenges and future research perspectives. In *First International Workshop on Privacy and Secuirty of Big Data*. ACM.

da Silva, R. F., Filgueira, R., Pietri, I., Jiang, M., Sakellariou, R., and Deelman, E. (2017). A characterization of workflow management systems for extreme-scale applications. *Future Generation Computer Systems*, 75:228–238.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Demchenko, Y., Turkmen, F., Slawik, M., and de Laat, C. (2017). Defining intercloud security framework and architecture components for multi-cloud data intensive applications. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 945–952. IEEE.

Di Martino, B., Cretella, G., and Esposito, A. (2015). Cross-platform cloud apis. In *Cloud Portability and Interoperability*, pages 45–57. Springer.

Dong, C., Wang, Y., Aldweesh, A., McCorry, P., and van Moorsel, A. (2017). Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. In *2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 211–227. ACM.

Dong, Y., Ye, W., Jiang, Y., Pratt, I., Ma, S., Li, J., and Guan, H. (2013). Colo: Coarse-grained lock-stepping virtual machines for non-stop service. In *4th annual Symposium on Cloud Computing*, page 3. ACM.

Dyer, D. (2010). Watchmaker framework for evolutionary computation.

Ebrahimi, M., Mohan, A., Lu, S., and Reynolds, R. (2015). Tps: A task placement strategy for big data workflows. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 523–530. IEEE.

Eldawy, A. and Mokbel, M. F. (2015). Spatialhadoop: A mapreduce framework for spatial data. In *2015 IEEE 31st international conference on Data Engineering*, pages 1352–1363. IEEE.

Fang, W., Wen, X. Z., Zheng, Y., and Zhou, M. (2017). A survey of big data security and privacy preserving. *IETE Technical Review*, 34(5):544–560.

Fernando, T., Gureev, N., Matskin, M., Zwick, M., and Natschläger, T. (2018). Workflowdsl: Scalable workflow execution with provenance for data analysis applications. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 774–779. IEEE.

Filgueira, R., da Silva, R. F., Krause, A., Deelman, E., and Atkinson, M. (2016). Asterism: Pegasus and dispel4py hybrid workflows for data-intensive science. In *2016 Seventh International Workshop on Data-Intensive Computing in the Clouds (DataCloud)*, pages 1–8. IEEE.

Filgueira, R., Krause, A., Atkinson, M., Klampanos, I., Spinuso, A., and Sanchez-Exposito, S. (2015). dispel4py: An agile framework for data-intensive escience. In *2015 IEEE 11th International Conference on e-Science*, pages 454–464. IEEE.

Filguiera, R., Krause, A., Atkinson, M., Klampanos, I., and Moreno, A. (2017). dispel4py: A python framework for data-intensive scientific computing. *The International Journal of High Performance Computing Applications*, 31(4):316–334.

Fischer, L. and Bernstein, A. (2015). Workload scheduling in distributed stream processors using graph partitioning. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 124–133. IEEE.

Foster, H., Uchitel, S., Magee, J., and Kramer, J. (2006). Model-based analysis of obligations in web service choreography. In *Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services (AICT-ICIW'06)*, pages 149–149. IEEE.

Fu, W.-T. and Dong, W. (2012). Collaborative indexing and knowledge exploration: A social learning model. *IEEE Intelligent Systems*, 27(1):39–46.

Gacto, M. J., Alcalá, R., and Herrera, F. (2010). Integration of an index to preserve the semantic interpretability in the multiobjective evolutionary rule selection and tuning of linguistic fuzzy systems. *IEEE Transactions on Fuzzy Systems*, 18(3):515–531.

Gajecka, M. (2016). Unrevealed mosaicism in the next-generation sequencing era. *Molecular Genetics and Genomics*, 291(2):513–530.

Gani, A., Siddiqa, A., Shamshirband, S., and Hanum, F. (2016). A survey on indexing techniques for big data: taxonomy and performance evaluation. *Knowledge and information systems*, 46(2):241–284.

Garcês, R., de Jesus, T., Cardoso, J., and Valente, P. (2009). Open source workflow management systems: A concise survey. *Chapter in Book*, pages 179–190.

García-Martínez, C., Rodriguez, F. J., and Lozano, M. (2018). *Genetic Algorithms*. Springer International Publishing.

Garg, S., Wang, S., and Ranjan, R. (2018). *Orchestration Tools for Big Data*, pages 1–9. Springer International Publishing.

Garg, S. K. and Buyya, R. (2011). Networkcloudsim: Modelling parallel applications in cloud simulations. In *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, pages 105–113. IEEE.

Ge, Y., Liang, X., Zhou, Y. C., Pan, Z., Zhao, G. T., and Zheng, Y. L. (2016). Adaptive analytic service for real-time internet of things applications. In *2016 IEEE International Conference on Web Services (ICWS)*, pages 484–491. IEEE.

Gerlach, W., Tang, W., Keegan, K., Harrison, T., Wilke, A., Bischof, J., D'Souza, M., Devoid, S., Murphy-Olson, D., Desai, N., et al. (2014). Skyport: container-based execution environment management for multi-cloud scientific workflows. In *5th International Workshop on Data-Intensive Computing in the Clouds*, pages 25–32. IEEE Press.

Giaglis, G. M. (2001). A taxonomy of business process modeling and information systems modeling techniques. *International Journal of Flexible Manufacturing Systems*, 13(2):209–228.

Glavic, B. (2014). Big data provenance: Challenges and implications for benchmarking. In *Specifying big data benchmarks*, pages 72–80. Springer.

Glavic, B., Esmaili, K. S., Fischer, P. M., and Tatbul, N. (2011). The case for fine-grained stream provenance. In *BTW Workshops*, volume 11.

Glavic, B., Esmaili, K. S., Fischer, P. M., and Tatbul, N. (2014). Efficient stream provenance via operator instrumentation. *ACM Transactions on Internet Technology (TOIT)*, 14(1):7.

Gomes, J., Bagnaschi, E., Campos, I., David, M., Alves, L., Martins, J., Pina, J., López-García, A., and Orviz, P. (2018). Enabling rootless linux containers in multi-user environments: the udocker tool. *Computer Physics Communications*, 232:84–97.

Gonidis, F., Simons, A. J., Paraskakis, I., and Kourtesis, D. (2013). Cloud application portability: an initial view. In *6th Balkan Conference in Informatics*, pages 275–282. ACM.

Google (2017). Google compute engine pricing.

Gouglidis, A. and Mavridis, I. (2009). On the definition of access control requirements for grid and cloud computing systems. In *International Conference on Networks for Grid Applications*, pages 19–26. Springer.

Hammoud, S., Li, M., Liu, Y., Alham, N. K., and Liu, Z. (2010). Mrsim: A discrete event based mapreduce simulator. In *2010 Seventh International Conference on Fuzzy Systems and Knowledge Discovery*, volume 6, pages 2993–2997. IEEE.

Harrer, S., Lenhard, J., and Wirtz, G. (2012). Bpel conformance in open source engines. In *2012 Fifth IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–8. IEEE.

Harrer, S., Lenhard, J., and Wirtz, G. (2013). Open source versus proprietary software in service-orientation: the case of bpel engines. In *International Conference on Service-Oriented Computing*, pages 99–113. Springer.

Harrer, S., Preißinger, C., and Wirtz, G. (2014). Bpel conformance in open source engines: the case of static analysis. In *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*, pages 33–40. IEEE.

Hassan, Q. F., Elkhodr, M., and Shahrestani, S. (2017a). *Networks of the Future: Architectures, Technologies, and Implementations*. Chapman and Hall/CRC.

Hassan, Q. F., Khan, A. u. R., and Madani, S. A. (2017b). *Internet of things: Challenges, Advances, and Applications*. Chapman and Hall/CRC.

He, D., Kumar, N., Wang, H., Wang, L., Choo, K.-K. R., and Vinel, A. (2018). A provably-secure cross-domain handshake scheme with symptoms-matching for mobile healthcare social network. *IEEE Transactions on Dependable and Secure Computing*, 15(4):633–645.

He, D., Zeadally, S., Kumar, N., and Wu, W. (2016). Efficient and anonymous mobile user authentication protocol using self-certified public key cryptography for multi-server architectures. *IEEE Transactions on Information Forensics and Security*, 11(9):2052–2064.

Higashino, W. A., Capretz, M. A., and Bittencourt, L. F. (2016). Cepsim: Modelling and simulation of complex event processing systems in cloud environments. *Future Generation Computer Systems*, 65:122–139.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., and Stoica, I. (2011). Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22.

Hirzel, M., Andrade, H., Gedik, B., Jacques-Silva, G., Khandekar, R., Kumar, V., Mendell, M., Nasgaard, H., Schneider, S., Soulé, R., et al. (2013). Ibm streams processing language: analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7–1.

Höfer, C. and Karagiannis, G. (2011). Cloud computing services: taxonomy and comparison. *Journal of Internet Services and Applications*, 2(2):81–94.

Hu, H., Wen, Y., Chua, T.-S., and Li, X. (2014). Toward scalable systems for big data analytics: A technology tutorial. *IEEE Access*, 2:652–687.

Hu, Z., Li, B., and Luo, J. (2016). Flutter: Scheduling tasks closer to data across geo-distributed datacenters. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE.

Hung, C.-C., Golubchik, L., and Yu, M. (2015). Scheduling jobs across geo-distributed datacenters. In *Sixth ACM Symposium on Cloud Computing*, pages 111–124. ACM.

Huq, M. R., Wombacher, A., and Apers, P. M. (2011). Inferring fine-grained data provenance in stream data processing: reduced storage cost, high accuracy. In *International Conference on Database and Expert Systems Applications*, pages 118–127. Springer.

Interlandi, M. and Condie, T. (2018). Supporting data provenance in data-intensive scalable computing systems. *IEEE Data Eng. Bull.*, 41(1):63–73.

Interlandi, M., Ekmekji, A., Shah, K., Gulzar, M. A., Tetali, S. D., Kim, M., Millstein, T., and Condie, T. (2018). Adding data provenance support to apache spark. *The VLDB JournalThe International Journal on Very Large Data Bases*, 27(5):595–615.

Isard, M. and Abadi, M. (2015). Falkirk wheel: Rollback recovery for dataflow systems. *arXiv preprint arXiv:1503.08877*.

Javadi, B., Thulasiramy, R. K., and Buyya, R. (2011). Statistical modeling of spot instance prices in public cloud environments. In *2011 fourth IEEE international conference on utility and cloud computing*, pages 219–228. IEEE.

Jin, Y., Gao, Y., Qian, Z., Zhai, M., Peng, H., and Lu, S. (2016). Workload-aware scheduling across geo-distributed data centers. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 1455–1462. IEEE.

Jrad, F., Tao, J., and Streit, A. (2012). Sla based service brokering in intercloud environments. *CLOSER*, 2012:76–81.

Jrad, F., Tao, J., and Streit, A. (2013). A broker-based framework for multi-cloud workflows. In *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*, pages 61–68. ACM.

Jung, J. and Kim, H. (2012). Mr-cloudsim: Designing and implementing mapreduce computing model on cloudsim. In *2012 International Conference on ICT Convergence (ICTC)*, pages 504–509. IEEE.

Kashlev, A. and Lu, S. (2014). A system architecture for running big data workflows in the cloud. In *2014 IEEE International Conference on Services Computing*, pages 51–58. IEEE.

Kaur, K., Dhand, T., Kumar, N., and Zeadally, S. (2017). Container-as-a-service at the edge: Trade-off between energy efficiency and service availability at fog nano data centers. *IEEE wireless communications*, 24(3):48–56.

Keenan, T. (2016). Streaming data: Big data at high velocity.

Khandekar, R., Hildrum, K., Parekh, S., Rajan, D., Wolf, J., Wu, K.-L., Andrade, H., and Gedik, B. (2009). Cola: Optimizing stream processing applications via graph partitioning. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 308–327. Springer.

Kiran, M., Murphy, P., Monga, I., Dugan, J., and Baveja, S. S. (2015). Lambda architecture for cost-effective batch and speed big data processing. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 2785–2792. IEEE.

Kombi, R. K., Lumineau, N., Lamarre, P., Rivetti, N., and Busnel, Y. (2019). Dabs-storm: A data-aware approach for elastic stream processing. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XL*, pages 58–93. Springer.

Komkhao, M., Lu, J., Li, Z., and Halang, W. A. (2013). Incremental collaborative filtering based on mahalanobis distance and fuzzy membership for recommender systems. *International Journal of General Systems*, 42(1):41–66.

Krutz, R. L., Vines, R. D., and Brunette, G. (2010). *Cloud security: A comprehensive guide to secure cloud computing*. Wiley Indianapolis.

Kurtzer, G. M., Sochat, V., and Bauer, M. W. (2017). Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5):e0177459.

Lama, P. and Zhou, X. (2012). Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *9th international conference on Autonomic computing*, pages 63–72. ACM.

Li, Z., Liu, L., and Tong, Z. (2017). Study on fault tolerance method in cloud platform based on workload consolidation model of virtual machine. *Journal of Engineering Science & Technology Review*, 10(5).

Lin, W., Qian, Z., Xu, J., Yang, S., Zhou, J., and Zhou, L. (2016). Streamscope: continuous reliable distributed processing of big data streams. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 439–453.

Liu, C., Yang, C., Zhang, X., and Chen, J. (2015a). External integrity verification for outsourced big data in cloud and iot: A big picture. *Future generation computer systems*, 49:58–67.

Liu, J., Li, J., Li, W., and Wu, J. (2016a). Rethinking big data: A review on the data quality and usage issues. *ISPRS Journal of Photogrammetry and Remote Sensing*, 115:134–142.

Liu, J., Pacitti, E., and Valduriez, P. (2018). A survey of scheduling frameworks in big data systems. *International Journal of Cloud Computing*, pages 1–27.

Liu, J., Pacitti, E., Valduriez, P., and Mattoso, M. (2015b). A survey of data-intensive scientific workflow management. *Journal of Grid Computing*, 13(4):457–493.

Liu, J., Silva, V., Pacitti, E., Valduriez, P., and Mattoso, M. (2014). Scientific workflow partitioning in multisite cloud. In *European Conference on Parallel Processing*, pages 105–116. Springer.

Liu, X. and Buyya, R. (2017). D-storm: Dynamic resource-efficient scheduling of stream processing applications. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 485–492. IEEE.

Liu, Y., Shi, X., and Jin, H. (2016b). Runtime-aware adaptive scheduling in stream processing. *Concurrency and Computation: Practice and Experience*, 28(14):3830–3843.

Liu, Y. and Wei, W. (2015). A replication-based mechanism for fault tolerance in mapreduce framework. *Mathematical Problems in Engineering*, 2015.

lKempf, R. (2017). Open source data pipeline luigi vs azkaban vs oozie vs airflow. `https://www.bizety.com/2017/06/05/open-source-data-pipeline-luigi-vs-azkaban-vs-oozie-vs-airflow/`.

Lopez, M. A., Lobato, A. G. P., and Duarte, O. C. M. (2016). A performance comparison of open-source stream processing platforms. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE.

Lozano, M., Herrera, F., and Cano, J. R. (2008). Replacement strategies to preserve useful diversity in steady-state genetic algorithms. *Information Sciences*, 178(23):4421–4433.

Ludäscher, B., Weske, M., McPhillips, T., and Bowers, S. (2009). Scientific workflows: Business as usual? In *International Conference on Business Process Management*, pages 31–47. Springer.

Lynn, D. (2016). Apache spark cluster managers: Yarn, mesos, or standalone? `http://www.agildata.com/apache-spark-cluster-managers-yarn-mesos-or-standalone/`.

Ma, Y., Rao, J., Hu, W., Meng, X., Han, X., Zhang, Y., Chai, Y., and Liu, C. (2012). An efficient index for massive iot data in cloud environment. In *21st ACM international conference on Information and knowledge management*, pages 2129–2133. ACM.

Mace, J. C., Van Moorsel, A., and Watson, P. (2011). The case for dynamic security solutions in public cloud workflow deployments. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*.

Malik, T., Nistor, L., and Gehani, A. (2010). Tracking and sketching distributed data provenance. In *2010 IEEE Sixth International Conference on e-Science*, pages 190–197. IEEE.

Mans, R., Van Der Aalst, W., Russell, N., and Bakker, P. (2009). Implementation of a healthcare process in four different workflow systems. *Technical report, Eindhoven, Netherlands: Technische Universiteit Eindhoven*.

Mansouri, Y., Toosi, A. N., and Buyya, R. (2017). Data storage management in cloud envirn.: Taxonomy, survey, and future directions. *ACM Computing Surveys (CSUR)*, 50(6):91.

Martello, S. (1981). An algorithm for the generalized assignment problem. *Operational research*, pages 589–603.

Matri, P., Costan, A., Antoniu, G., Montes, J., and Pérez, M. (2016). *Tỳr: Efficient Transactional Storage for Data-Intensive Applications*. PhD thesis, Inria Rennes Bretagne Atlantique; Universidad Politécnica de Madrid.

Mattsson, U. (2016). Data centric security key to cloud and digital business.

Meyer, D. (2014). Data analytics in the lab. `http://www.clpmag.com/2014/07/data-analytics-lab/`.

Microsoft (2017). Windows virtual machines pricing. `https://azure.microsoft.com/en-au/pricing/details/virtual-machines/windows/`.

Mikami, S., Ohta, K., and Tatebe, O. (2011). Using the gfarm file system as a posix compatible storage platform for hadoop mapreduce applications. In *2011 IEEE/ACM 12th International Conference on Grid Computing*, pages 181–189. IEEE.

Mohammadi, M., Al-Fuqaha, A., Sorour, S., and Guizani, M. (2018). Deep learning for iot big data and streaming analytics: A survey. *IEEE Communications Surveys & Tutorials*, 20(4):2923–2960.

Mohan, A., Ebrahimi, M., Lu, S., and Kotov, A. (2016). A nosql data model for scalable big data workflow execution. In *2016 IEEE International Congress on Big Data (BigData Congress)*, pages 52–59. IEEE.

Mon, E. E., Thein, M. M., and Aung, M. T. (2016). Clustering based on task dependency for data-intensive workflow scheduling optimization. In *2016 9th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS)*, pages 20–25. IEEE.

Murthy, A. C. (2009). Mumak: Map-reduce simulator. *MAPREDUCE-728, Apache JIRA*.

Nachiappan, R., Javadi, B., Calheiros, R. N., and Matawie, K. M. (2017). Cloud storage reliability for big data applications: A state of the art survey. *Journal of Network and Computer Applications*, 97:35–47.

Pandey, S. and Buyya, R. (2012). A survey of scheduling and management techniques for data-intensive application workflows. In *Data Intensive Distributed Computing: Challenges and Solutions for Large-scale Information Management*, pages 156–176. IGI Global.

Park, H., Ikeda, R., and Widom, J. (2011). Ramp: A system for capturing and tracing provenance in mapreduce workflows. *Stanford InfoLab*.

Pawluk, P., Simmons, B., Smit, M., Litoiu, M., and Mankovski, S. (2012). Introducing stratos: A cloud broker service. In *2012 IEEE fifth international conference on cloud computing*, pages 891–898. IEEE.

Peoples, C., Parr, G., Oredope, A., and Moessner, K. (2013). The standardisation of cloud computing: Trends in the state-of-the-art and management issues for the next generation of cloud. In *2013 Science and Information Conference*, pages 902–911. IEEE.

Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., and Seltzer, M. (2006). Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 49–49. IEEE.

Poola, D., Ramamohanarao, K., and Buyya, R. (2014). Fault-tolerant workflow scheduling using spot instances on clouds. *Procedia Computer Science*, 29:523–533.

Poola, D., Ramamohanarao, K., and Buyya, R. (2016). Enhancing reliability of workflow execution using task replication and spot instances. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(4):30.

Poola, D., Salehi, M., Ramamohanarao, K., and Buyya, R. (2017). *Chapter 15 - A Taxonomy and Survey of Fault-Tolerant Workflow Management Systems in Cloud and Distributed Computing Environments*. Morgan Kaufmann.

Postscapes (2017). Iot standards and protocols. `https://www.postscapes.com/internet-of-things-protocols/`.

Qasha, R., Cala, J., and Watson, P. (2016). Dynamic deployment of scientific workflows in the cloud using container virtualization. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 269–276. IEEE.

Rahman, M., Ranjan, R., Buyya, R., and Benatallah, B. (2011). A taxonomy and survey on autonomic management of applications in grid computing environments. *Concurrency and computation: practice and experience*, 23(16):1990–2019.

Ranjan, R., Garg, S., Khoskbar, A. R., Solaiman, E., James, P., and Georgakopoulos, D. (2017). Orchestrating bigdata analysis workflows. *IEEE Cloud Computing*, 4(3):20–28.

Ranjan, R., Kolodziej, J., Wang, L., and Zomaya, A. Y. (2015). Cross-layer cloud resource configuration selection in the big data era. *IEEE Cloud Computing*, 2(3):16–22.

Rao, T. R., Mitra, P., Bhatt, R., and Goswami, A. (2018). The big data system, components, tools, and technologies: a survey. *Knowledge and Information Systems*, pages 1–81.

Reddy, K. H. K. and Roy, D. S. (2015). Dppacs: A novel data partitioning and placement aware computation scheduling scheme for data-intensive cloud applications. *The Computer Journal*, 59(1):64–82.

Redlich, D., Blair, G., Rashid, A., Molka, T., and Gilani, W. (2014). Research challenges for business process models at run-time. In *Models@ run. time*, pages 208–236. Springer.

Rehani, N. and Garg, R. (2017). Reliability-aware workflow scheduling using monte carlo failure estimation in cloud. In *International Conference on Communication and Networks*, pages 139–153. Springer.

Reijers, H. A. (2003). *Design and control of workflow processes: business process management for the service industry*. Springer-Verlag.

Rizou, S., Durr, F., and Rothermel, K. (2010). Solving the multi-operator placement problem in large-scale operator networks. In *2010 19th International Conference on Computer Communications and Networks*, pages 1–6. IEEE.

Rodriguez, M. A. and Buyya, R. (2017). A taxonomy and survey on scheduling algorithms for scientific workflows in iaas cloud computing environments. *Concurrency and Computation: Practice and Experience*, 29(8).

Rodríguez-García, M. Á., Valencia-García, R., García-Sánchez, F., and Samper-Zapater, J. J. (2014). Creating a semantically-enhanced cloud services environment through ontology evolution. *Future Generation Computer Systems*, 32:295–306.

Saha, B. and Srivastava, D. (2014). Data quality: The other face of big data. In *2014 IEEE 30th International Conference on Data Engineering*, pages 1294–1297. IEEE.

Sakr, S., Liu, A., Batista, D. M., and Alomari, M. (2011). A survey of large scale data management approaches in cloud environments. *IEEE Communications Surveys & Tutorials*, 13(3):311–336.

Sakr, S., Liu, A., and Fayoumi, A. G. (2013). The family of mapreduce and large-scale data processing systems. *ACM Computing Surveys (CSUR)*, 46(1):11.

Sansrimahachai, W., Moreau, L., and Weal, M. J. (2013). An on-the-fly provenance tracking mechanism for stream processing systems. In *2013 IEEE/ACIS 12th International Conference on Computer and Information Science (ICIS)*, pages 475–481. IEEE.

Scott, J. (2015). A tale of two clusters: Mesos and yarn. `https://www.oreilly.com/ideas/a-tale-of-two-clusters-mesos-and-yarn`.

Seiger, R., Huber, S., and Schlegel, T. (2018). Toward an execution system for self-healing workflows in cyber-physical systems. *Software & Systems Modeling*, 17(2):551–572.

Shishido, H. Y., Estrella, J. C., Toledo, C. F. M., and Reiff-Marganiec, S. (2018). (wip) tasks selection policies for securing sensitive data on workflow scheduling in clouds. In *2018 IEEE International Conference on Services Computing (SCC)*, pages 233–236. IEEE.

Shu, T. and Wu, C. Q. (2017). Performance optimization of hadoop workflows in public clouds through adaptive task partitioning. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE.

Silva, V., de Oliveira, D., Valduriez, P., and Mattoso, M. (2018). Dfanalyzer: runtime dataflow analysis of scientific applications using provenance. *VLDB Endowment*, 11(12):2082–2085.

Souza, A., Papadopoulos, A. V., Tomas, L., Gilbert, D., and Tordsson, J. (2018). Hybrid adaptive checkpointing for virtual machine fault tolerance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 12–22. IEEE.

Sowe, S. K., Kimata, T., Dong, M., and Zettsu, K. (2014). Managing heterogeneous sensor data on a big data platform: Iot services for data-intensive science. In *2014 IEEE 38th International Computer Software and Applications Conference Workshops*, pages 295–300. IEEE.

Sphere, M. (2017a). Apache mesos.

Sphere, M. (2017b). Apache mesos documentation.

Sun, D. and Huang, R. (2016). A stable online scheduling strategy for real-time stream computing over fluctuating big data streams. *IEEE Access*, 4:8593–8607.

Sun, D., Yan, H., Gao, S., Liu, X., and Buyya, R. (2018). Rethinking elastic online scheduling of big data streaming applications over high-velocity continuous data streams. *The Journal of Supercomputing*, 74(2):615–636.

Sun, D., Zhang, G., Wu, C., Li, K., and Zheng, W. (2017). Building a fault tolerant framework with deadline guarantee in big data stream computing environments. *Journal of Computer and System Sciences*, 89:4–23.

Sun, D., Zhang, G., Yang, S., Zheng, W., Khan, S. U., and Li, K. (2015). Re-stream: Real-time and energy-efficient resource scheduling in big data stream computing environments. *Information Sciences*, 319:92–112.

Talbi, E.-G., Basseur, M., Nebro, A. J., and Alba, E. (2012). Multi-objective optimization using metaheuristics: non-standard algorithms. *International Transactions in Operational Research*, 19(1-2):283–305.

Talia, D. (2013). Workflow systems for science: Concepts and tools. *ISRN Software Engineering*, 2013.

Tan, W., Tata, S., Tang, Y., and Fong, L. L. (2014). Diff-index: Differentiated index in distributed log-structured data stores. In *EDBT*, pages 700–711.

Taneja, M. and Davy, A. (2017). Resource aware placement of iot application modules in fog-cloud computing paradigm. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 1222–1228. IEEE.

Tang, H. (2009). Mumak: Map-reduce simulator. *Mumak: apache, 2009 (2009-09-25)[2011-11-26]*.

Teng, F., Yang, H., Li, T., Yang, Y., and Li, Z. (2013). Scheduling real-time workflow on mapreduce-based cloud. In *Third International Conference on Innovative Computing Technology (INTECH 2013)*, pages 117–122. IEEE.

Todd Jr, S., Baldwin, R., Dietrich, D., and Pauley Jr, W. A. (2017). Data analytics computing resource provisioning based on computed cost and time parameters for proposed computing resource configurations. US Patent 9,684,866.

Tönjes, R., Barnaghi, P., Ali, M., Mileo, A., Hauswirth, M., Ganz, F., Ganea, S., Kjærgaard, B., Kuemper, D., Nechifor, S., et al. (2014). Real time iot stream processing and large-scale data analytics for smart city applications. In *poster session, European Conference on Networks and Communications*. sn.

Toosi, A. N., Sinnott, R. O., and Buyya, R. (2018). Resource provisioning for data-intensive applications with deadline constraints on hybrid clouds using aneka. *Future Generation Computer Systems*, 79:765–775.

Tudoran, R., Costan, A., and Antoniu, G. (2016). Overflow: multi-site aware big data management for scientific workflows on clouds. *IEEE transactions on cloud computing*, 4(1):76–89.

Ulmer, C., Mukherjee, S., Templet, G., Levy, S., Lofstead, J., Widener, P., Kordenbrock, T., and Lawson, M. (2018). Faodel: Data management for next-generation application workflows. In *9th Workshop on Scientific Cloud Computing*, page 8. ACM.

Van Der Aalst, W. M. and Ter Hofstede, A. H. (2005). Yawl: yet another workflow language. *Information systems*, 30(4):245–275.

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. (2013). Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM.

Venkataraman, S., Panda, A., Ousterhout, K., Armbrust, M., Ghodsi, A., Franklin, M. J., Recht, B., and Stoica, I. (2017). Drizzle: Fast and adaptable stream processing at scale. In *26th Symposium on Operating Systems Principles*, pages 374–389. ACM.

Verma, A., Cherkasova, L., and Campbell, R. H. (2011). Play it again, simmr! In *2011 IEEE International Conference on Cluster Computing*, pages 253–261. IEEE.

Vijayakumar, N. and Plale, B. (2007). Tracking stream provenance in complex event processing systems for workflow-driven computing. In *EDA-PS Workshop*.

Vishwakarma, S. K., Lakhtaria, K. I., Bhatnagar, D., and Sharma, A. K. (2014). An efficient approach for inverted index pruning based on document relevance. In *2014 Fourth*

International Conference on Communication Systems and Network Technologies, pages 487–490. IEEE.

von Leon, D., Miori, L., Sanin, J., El Ioini, N., Helmer, S., and Pahl, C. (2019). A lightweight container middleware for edge cloud architectures. *Fog and edge computing: principles and paradigms*, pages 145–170.

Vrable, M., Savage, S., and Voelker, G. M. (2012). Bluesky: A cloud-backed file system for the enterprise. In *10th USENIX conference on File and Storage Technologies*, pages 19–19. USENIX Association.

Wang, G., Butt, A. R., Pandey, P., and Gupta, K. (2009a). Using realistic simulation for performance analysis of mapreduce setups. In *1st ACM workshop on Large-Scale system and application performance*, pages 19–26. ACM.

Wang, J., Crawl, D., and Altintas, I. (2009b). Kepler+ hadoop: a general architecture facilitating data-intensive applications in scientific workflow systems. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, page 12. ACM.

Wang, J., Crawl, D., Altintas, I., and Li, W. (2014a). Big data applications using workflows for data parallel computing. *Computing in Science & Engineering*, 16(4):11–21.

Wang, K., Qiao, K., Sadooghi, I., Zhou, X., Li, T., Lang, M., and Raicu, I. (2016). Load-balanced and locality-aware scheduling for data-intensive workloads at extreme scales. *Concurrency and Computation: Practice and Experience*, 28(1):70–94.

Wang, K., Zhou, X., Li, T., Zhao, D., Lang, M., and Raicu, I. (2014b). Optimizing load balancing and data-locality with data-aware scheduling. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 119–128. IEEE.

Wang, Y., Lu, P., and Kent, K. B. (2014c). Wafs: a workflow-aware file system for effective storage utilization in the cloud. *IEEE Transactions on Computers*, 64(9):2716–2729.

Wang, Y. and Shi, W. (2014). Budget-driven scheduling algorithms for batches of mapreduce jobs in heterogeneous clouds. *IEEE Transactions on Cloud Computing*, 2(3):306–319.

Wen, Z., Cała, J., Watson, P., and Romanovsky, A. (2017). Cost effective, reliable and secure workflow deployment over federated clouds. *IEEE TSC.*, 10(6):929–941.

Wu, F., Wu, Q., and Tan, Y. (2015). Workflow scheduling in cloud: a survey. *The Journal of Supercomputing*, 71(9):3373–3418.

Wu, K., Shoshani, A., and Stockinger, K. (2010). Analyses of multi-level and multi-component compressed bitmap indexes. *ACM Transactions on Database Systems (TODS)*, 35(1):2.

Xu, C., Holzemer, M., Kaul, M., Soto, J., and Markl, V. (2017). On fault tolerance for distributed iterative dataflow processing. *IEEE Transactions on Knowledge and Data Engineering*, 29(8):1709–1722.

Yang, S. (2008). Genetic algorithms with memory-and elitism-based immigrants in dynamic environments. *Evolutionary Computation*, 16(3):385–416.

Yıldırım, H., Chaoji, V., and Zaki, M. J. (2012). Grail: a scalable index for reachability queries in very large graphs. *The VLDB JournalThe International Journal on Very Large Data Bases*, 21(4):509–534.

Yu, J. and Buyya, R. (2005). A taxonomy of scientific workflow systems for grid computing. *ACM Sigmod Record*, 34(3):44–49.

Yu, Y., Zhu, Y., Ng, W., and Samsudin, J. (2014). An efficient multidimension metadata index and search system for cloud data. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 499–504. IEEE.

Yuan, D., Yang, Y., and Chen, J. (2012). *Computation and Storage in the Cloud: Understanding the Trade-offs*. Newnes.

Zaharia, M., Das, T., Li, H., Shenker, S., and Stoica, I. (2012). Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Presented as part of the*.

Zeng, X., Garg, S., Wen, Z., Strazdins, P., Wang, L., and Ranjan, R. (2016). Sla-aware scheduling of map-reduce applications on public clouds. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 655–662. IEEE.

Zeng, X., Garg, S. K., Strazdins, P., Jayaraman, P. P., Georgakopoulos, D., and Ranjan, R. (2017). Iotsim: A simulator for analysing iot applications. *Journal of Systems Architecture*, 72:93–107.

Zeng, X., Garg, S. K., Wen, Z., Strazdins, P., Zomaya, A. Y., and Ranjan, R. (2018). Cost efficient scheduling of mapreduce applications on public clouds. *Journal of computational science*, 26:375–388.

Zhang, Z., Wu, C., and Cheung, D. W. (2013). A survey on cloud interoperability: taxonomies, standards, and practice. *ACM SIGMETRICS Performance Evaluation Review*, 40(4):13–22.

Zhao, Q., Xiong, C., and Wang, P. (2016a). Heuristic data placement for data-intensive applications in heterogeneous cloud. *Journal of Electrical and Computer Engineering*, 2016.

Zhao, Q., Xiong, C., Yu, C., Zhang, C., and Zhao, X. (2016b). A new energy-aware task scheduling method for data-intensive applications in the cloud. *Journal of Network and Computer Applications*, 59:14–27.

Zhao, Q., Xiong, C., Zhao, X., Yu, C., and Xiao, J. (2015a). A data placement strategy for data-intensive scientific workflows in cloud. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 928–934. IEEE.

Zhao, Y., Calheiros, R. N., Gange, G., Ramamohanarao, K., and Buyya, R. (2015b). Sla-based resource scheduling for big data analytics as a service in cloud computing environments. In *2015 44th International Conference on Parallel Processing*, pages 510–519. IEEE.

Zhao, Y., Li, Y., Lu, S., Raicu, I., and Lin, C. (2014). Devising a cloud scientific workflow platform for big data. In *2014 IEEE World Congress on Services*, pages 393–401. IEEE.

Zhao, Y., Li, Y., Raicu, I., Lu, S., Tian, W., and Liu, H. (2015c). Enabling scalable scientific workflow management in the cloud. *Future Generation Computer Systems*, 46:3–16.

Zhao, Y., Raicu, I., and Foster, I. (2008). Scientific workflow systems for 21st century, new bottle or new wine? In *2008 IEEE Congress on Services-Part I*, pages 467–471. IEEE.

Zhao, Z., van Oudenaarde, P. v. H. B., Belloum, F. T. A., and Hertzberger, V. K. P. S. B. (2005). Including the state of the art scientific workflow management systems in an e-science environment. In *1st IEEE International Conference on e-Science and Grid Computing*.

Zheng, C. and Thain, D. (2015). Integrating containers into workflows: A case study using makeflow, work queue, and docker. In *8th International Workshop on Virtualization Technologies in Distributed Computing*, pages 31–38. ACM.

Zhou, C. and Garg, S. K. (2015). Performance analysis of scheduling algorithms for dynamic workflow applications. In *2015 IEEE International Congress on Big Data*, pages 222–229. IEEE.

Zhu, X., Wang, J., Guo, H., Zhu, D., Yang, L. T., and Liu, L. (2016). Fault-tolerant scheduling for real-time scientific workflows with elastic resource provisioning in virtualized clouds. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3501–3517.

Zomaya, A. Y. and Sakr, S. (2017). *Handbook of big data technologies*. Springer.